Installing lcc

Christopher W. Fraser
AT&T Bell Laboratories Rm. 2C−300, 600 Mountain Ave., Murray Hill, NJ 07974−0636

David R. Hanson
Department of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544

Contents

1. Introduction

`lcc` is the ANSI C compiler described in our book *A Retargetable C Compiler: Design and Implementation* (Benjamin/Cummings, 1995, ISBN 0−8053−1670−1).

Extract the distribution into its own directory. All paths below are relative to this directory. The distribution holds the following subdirectories.

`src`
: source code

`etc`
: driver, accessories

`cpp`
: preprocessor source code

`lburg`
: code−generator generator source code

`doc`
: this document, man pages

`include`
: ANSI include files

`tst`
: test suite

`mips/* mips/*/tst`
: MIPS build directories, test outputs

`sparc/* sparc/*/tst`
: SPARC build directories, test outputs

`x86/* x86/*/tst`
: x86 build directories, test outputs

Distributions include code generators for the MIPS, SPARC, and the Intel 386 and its successors.

Installation on Unix systems involves three steps performed in the following order.

1. Decide where to install the man page, the include files, the compiler, and `lcc`, the driver program; see Sec. 2.
2. Install a preprocessor; see Sec. 3.
3. Install a host−specific driver; see Sec. 4.
4. Install the compiler; see Sec. 5.

The value of the variable `rcsid` in `src/main.c` identifies the version of the distribution. If the file `LOG` appears, it describes the changes from the previous version.

If you're installing `lcc` on a PC, go to Sec. 6 after skimming Secs. 2−5.

`doc/install.html` is the HTML file for this document. `doc/install.ps` and `doc/install.txt` are PostScript and plain ASCII versions.

2. Paths

Installation consists of four files and one directory; these are summarized below along with paths used in typical installations.

`/usr/local/man/man1/lcc.1`
: the man page

`/usr/local/bin/lcc`
: the driver

`/usr/local/lib/cpp`
: the preprocessor

`/usr/local/lib/rcc`
: the compiler

`/usr/local/include/ansi`
: include files (a directory)

These files can be placed in other, site−specific locations, but the compiler should be named `rcc`. If the driver isn't named `lcc`, edit the man page (`doc/lcc.1`).

Include files are in directories named `include/`*target*`/`*system*; the meaningful combinations are

`mips/irix`
: IRIX Release 4.0

`mips/ultrix`
: ULTRIX 4.3

`sparc/sun`
: SunOS 4.1

`sparc/solaris`
: Solaris 2.3

`x86/dos`
: DOS 6.0, Windows 3.1

Choose the include files that are appropriate for your system, or make a copy of a closely related set and edit them.

For example, if the paths shown above are chosen and if `include/mips/ultrix` has the appropriate include files, install the man page and include files by

```
% cp doc/lcc.1 /usr/local/man/man1
% cp include/mips/ultrix/*.h /usr/local/include/ansi
```

3. Installing the Preprocessor

`lcc` is intended to be used with an ANSI preprocessor, like the GNU C preprocessor available from the Free Software Foundation. If you have the GNU C preprocessor, you can use it and skip the reset of this section. You'll need the preprocessor's location to configure the driver, as described in the next section.

The directory `cpp` contains the source code for an ANSI preprocessor written by Dennis Ritchie. It's written in ANSI C, so you must use an ANSI C compiler to compile it. Alternatively, you can install `lcc` with an existing preprocessor, use `lcc` to build and install `cpp`, and then reconfigure the driver to use the new `cpp`.

To build and install `cpp`, execute the commands

```
% cd cpp
% make
% cp cpp /usr/local/lib
```

where the destination is the location chosen for `cpp` in Sec. 2. Use the `CC=` option to specify an ANSI C compiler, if necessary; e.g., `CC=gcc`. The command "`make clean`" cleans up, but does not remove `cpp`, and "`make clobber`" cleans up and removes `cpp`.

This preprocessor makes *no* assumptions about the standard include files. You must specify the appropriate `-I` options in the `include` array defined in the host-specific part of the driver, as detailed in the next section.

4. Installing the Driver

The preprocessor, compiler, assembler, and loader are invoked by a driver program, `lcc`, which is similar to `cc` on most systems. It's described in the man page `doc/lcc.1`. The driver is built by combining the host-independent part, `etc/lcc.c`, with a small host-specific part. By convention, host-specific parts are named *hostname*`.c`, where *hostname* is the local name for the host on which `lcc` is being installed. `etc` holds many examples. Comments in most give the details of the particular host; pick one that is closely related to your host, copy it to `etc/`*yourhostname*`.c`, and edit it as described below. You should not have to edit `etc/lcc.c`.

Debug your version of the driver by running it with the `-v -v` options, which cause it to echo the commands it would execute, but not to execute them.

Here's `etc/hart.c`, which we'll use as an example in describing how to edit a host-specific part. This example illustrates all the important features.

```
/* DECStations running ULTRIX at Princeton University */

#include <string.h>

char *cpp[] = {
        "/usr/gnu/lib/gcc-cpp", "-undef",
        "-DLANGUAGE_C", "-D_LANGUAGE_C", "-D__LANGUAGE_C",
        "-D_unix", "-D__unix", "-Dultrix", "-D_ultrix", "-D__ultrix",
        "-Dmips", "-D_mips", "-D__mips",
        "-Dhost_mips", "-D_host_mips", "-D__host_mips",
        "-DMIPSEL", "-D_MIPSEL", "-D__MIPSEL",
        "$1", "$2", "$3", 0 };
```

```
char *include[] = { "-I/usr/local/include/ansi", 0 };
char *com[] =  { "/usr/local/lib/rcc", "-target=mips-ultrix",
        "$1", "$2", "$3", 0 };
char *as[] =  { "/bin/as", "-o", "$3", "", "$1",
        "-nocpp", "-EL", "$2", 0 };
char *ld[] =  { "/usr/bin/ld", "-o", "$3", "/usr/lib/crt0.o",
        "$1", "$2", "", "", "-lm", "-lc", 0 };

int option(arg) char *arg; {
        if (strcmp(arg, "-g") == 0)
                as[3] = "-g";
        else if (strcmp(arg, "-p") == 0
        && strcmp(ld[3], "/usr/lib/crt0.o") == 0) {
                ld[3] = "/usr/lib/mcrt0.o";
                ld[7] = "/usr/lib/libprof1.a";
        } else if (strcmp(arg, "-b") == 0
        && access("/usr/local/lib/bbexit.o", 4) == 0)
                ld[6] = "/usr/local/lib/bbexit.o";
        else
                return 0;
        return 1;
}
```

Most of the host–specific code is data that gives prototypes for the commands that invoke the preprocessor, compiler, assembler, and loader. Each command prototype is an array of pointers to strings terminated with a null pointer; the first string is the full path name of the command and the others are the arguments or argument placeholders, which are described below.

The `cpp` array gives the command for running the preprocessor. `lcc` is intended to be used with an ANSI preprocessor, like the GNU C preprocessor. If the GNU C preprocessor is used, as shown in this example, it must be named `gcc-cpp` in order for `lcc`'s `-N` option to work correctly.

Literal arguments specified in prototypes, e.g., `"-Dmips"` in the `cpp` command above, are passed to the command as given.

The strings `"$1"`, `"$2"`, and `"$3"` in prototypes are placeholders for *lists* of arguments that are substituted in a copy of the prototype before the command is executed. `$1` is replaced by the *options* specified by the user; for the preprocessor, this list always contains at least `-Dunix` and `-D__LCC__`. `$2` is replaced by the *input* files, and `$3` is replaced by the *output* file.

Zero–length arguments after replacement are removed from the argument list before the command is invoked. So, e.g., if the preprocessor is invoked without an output file, `"$3"` becomes `""`, which is removed from the final argument list.

For example, to specify a preprocessor command prototype to invoke the preprocessor installed in Sec. 3 with the options `-Dmips` and `-Dultrix`, the `cpp` array would be

```
char *cpp[] = { "/usr/local/lib/cpp", "-Dmips", "-Dultrix",
        "$1", "$2", "$3", 0 };
```

where `cpp[0]` is the location chosen for `cpp` in Sec. 2.

The `include` array is a list of `-I` options that specify which directives should be searched to satisfy include directives. These directories are searched in the order given. The first directory should be the one to which the ANSI header files were copied in Sec. 2. The driver adds these options to `cpp`'s arguments when it invokes the preprocessor, except when `-N` is specified.

Design this list carefully. Mixing ANSI and pre–ANSI headers (e.g., by listing `/usr/include` after the directory of ANSI headers, as shown above) may mix incompatible headers. Unless the default list holds *only* `/usr/include` or *only* the ANSI headers, many users may be forced to use `-N` and `-I` incessantly.

`com` gives the command for invoking the compiler. This prototype can appear as shown above, with two important changes. The command name should be edited to reflect the location of the compiler chosen in Sec. 2, and the option `-target=mips-ultrix` should be edited to the *target-system* for your host. `lcc` can generate code for *all* of the *target-system* combinations listed in the file `src/bind.c`. The `-target` option specifies the default combination. The driver's `-Wf` option can be used to specify other combinations; the man page elaborates.

`as` gives the command for invoking the assembler.

`ld` gives the command for invoking the loader. For the other commands, the list `$2` contains a single file; for `ld`, `$2` contains all ".o" files and libraries, and `$3` is `a.out`, unless the `-o` option is specified. As suggested in the code above, `ld` must also specify the appropriate startup code and default libraries.

The `option` function is described below; for now, use an existing `option` function or one that returns 0.

After specifying the prototypes, compile the driver by

```
% cd etc
% make HOST=hart
cc  -c hart.c
cc  -c lcc.c
cc -s hart.o lcc.o; rm -f hart.o lcc.o
```

where `hart` is replaced by *yourhostname*. Run the resulting `a.out` with the options `-v -v` to display the commands that would be executed, e.g.,

```
% a.out -v -v foo.c baz.c mylib.a -lX11
a.out $ Revision: 3.2 $ $ Date: 1994/09/08 17:15:13 $
foo.c:
/usr/gnu/lib/gcc-cpp -undef -DLANGUAGE_C -D_LANGUAGE_C -D__LANGUAGE_C
   -D_unix -D__unix -Dultrix -D_ultrix -D__ultrix -Dmips -D_mips -D__mips
   -Dhost_mips -D_host_mips -D__host_mips -DMIPSEL -D_MIPSEL -D__MIPSEL
   -Dunix -D__LCC__ -v -I/usr/local/include/ansi foo.c |
   /usr/local/lib/rcc -target=mips-ultrix -v - /tmp/lcc11717.s
/bin/as -o foo.o -nocpp -EL /tmp/lcc11717.s
baz.c:
/usr/gnu/lib/gcc-cpp -undef -DLANGUAGE_C -D_LANGUAGE_C -D__LANGUAGE_C
   -D_unix -D__unix -Dultrix -D_ultrix -D__ultrix -Dmips -D_mips -D__mips
   -Dhost_mips -D_host_mips -D__host_mips -DMIPSEL -D_MIPSEL -D__MIPSEL
   -Dunix -D__LCC__ -v -I/usr/local/include/ansi baz.c |
   /usr/local/lib/rcc -target=mips-ultrix -v - /tmp/lcc11717.s
/bin/as -o baz.o -nocpp -EL /tmp/lcc11717.s
/usr/bin/ld -o a.out /usr/lib/crt0.o foo.o baz.o mylib.a -lX11 -lm -lc
rm /tmp/lcc11717.s
```

Leading spaces indicate lines that have been folded manually to fit this page. Note the use of a pipeline to connect the preprocessor and compiler. `lcc` arranges this pipeline itself; it does not call the shell. If you want `lcc` to use temporary files instead of a pipeline, define `PIPE=0` in `CFLAGS` when making the driver:

```
% make CFLAGS='-DPIPE=0' HOST=hart
```

The option `-pipe` forces `lcc` to use a pipeline between the preprocessor and the compiler regardless of `PIPE`'s value.

As the output shows, `lcc` places temporary files in `/tmp`. Alternatives can be specified by defining `TEMPDIR` in `CFLAGS` when making the driver, e.g.,

```
% make CFLAGS='-DTEMPDIR=\"/usr/tmp\"' HOST=hart
```

causes `lcc` to place temporary files in `/usr/tmp`. Once the driver is completed, install it by

```
% cp a.out /usr/local/bin/lcc
```

where the destination is the location chosen for `lcc` in Sec. 2.

The `option` function is called for the options `-g`, `-p`, `-pg`, and `-b` because these compiler options might also affect the loader's arguments. For these options, the driver calls `option(arg)` to give the host–specific code an opportunity to edit the `ld` command, if necessary. `option` can change `ld`, if necessary, and return 1 to announce its acceptance of the option. If the option is unsupported, `option` should return 0.

For example, in response to `-g`, the `option` function shown above changes `as[3]` from `""` to `"-g"`, which specifies the debugging option to the assembler. If `-g` is not specified, the `""` argument is omitted from the `as` command because it's empty.

Likewise, the `-p` causes `option` to change the name of the startup code and add the name of the profiling library. Note that `option` has been written to support simultaneous use of `-g` and `-p`, e.g.,

```
% a.out -v -v -g -p foo.s baz.o -o myfoo
a.out $ Revision: 3.2 $ $ Date: 1994/09/08 17:15:13 $
/bin/as -o foo.o -g -nocpp -EL foo.s
/usr/bin/ld -o myfoo /usr/lib/mcrt0.o foo.o baz.o
   /usr/lib/libprof1.a -lm -lc
rm /tmp/lcc12270.s
```

On Suns, the driver also recognizes `-Bstatic` and `-Bdynamic` as linker options, and recognizes but ignores Sun's "`-target` *name*" option.

The option `-Wo`*arg* causes the driver to pass *arg* to `option`. Such options have no other effect; this mechanism is provided to support system–specific options that affect the commands executed by the driver.

To complete the driver, write an appropriate `option` function for your system, and make and install the driver as described above.

4.1 Installing bprint

The `-b` option causes the compiler to generate code to count the number of times each expression is executed. The `exit` function in `etc/bbexit.c` writes these counts to `prof.out` when the program terminates. If `option` is called with `-b`, it must edit the `ld` command accordingly, as shown above. This version of `option` uses the `access` system call to insure that `bbexit.o` is installed before editing the `ld` command. To install `bbexit.o` execute

```
% make bbexit.o
% cp bbexit.o /usr/local/lib/bbexit.o
```

If necessary, change `/usr/local/lib` to reflect local conventions. The `exit` function in `etc/bbexit.c` works on the systems listed in Sec. 2, but may need to be modified for other systems.

If `option` supports `-b`, you should also install `etc/bprint.c`, which reads `prof.out` and generates a listing annotated with execution counts. After `lcc` is installed, install `bprint` with the commands

```
% make bprint
% cp bprint /usr/local/bin/bprint
% cp ../doc/bprint.1 /usr/local/man/man1
```

The `makefile` uses `lcc` to compile `bprint.c`; you must use `lcc` or another ANSI C compiler, e.g., `gcc`, because `bprint.c` is written in ANSI C. Also, `bprint.c` *includes* `"../src/profio.c"`, so it must be compiled in `etc`.

5. Installing the Compiler Proper

The compiler proper, `rcc`, is built by compiling it with the host C compiler and then using the result to re−compile itself. A test suite is used to verify that the compiler is working correctly. The examples below illustrate this process on a MIPS under Ultrix. You must have the driver, `lcc`, installed in order to test `rcc`. If any of the steps below fail, contact us (see Sec. 7).

The object files, `rcc`, and the generated code for the programs in the test suite are placed in the directory *target/system* where *target* and *system* are the names of your target machine and its operating system, respectively. There are directories for the supported *target/system* combinations, e.g., `mips/ultrix`.

The default target in `src/makefile` is `rcc`. `lcc` is built by executing `make` from the apppropriate *target/system* directory and specifying system−specific values for `CFLAGS` and `LDFLAGS`, if necessary. For example, to build `rcc` for a MIPS running Ultrix, execute the commands

```
% cd mips/ultrix
% make -f ../../src/makefile
cc -c -O ../../src/alloc.c
...
cc -c -O ../../src/x86.c
cc -o rcc  alloc.o bind.o ... mips.o sparc.o x86.o
```

There may be a few warnings, but there should be no errors. If your host is an SGI machine running IRIX 4.0 or later, you might need `CFLAGS=-cckr`. If `cc` doesn’t automatically search the directory that holds the source file, specify `CFLAGS=-I../../src`. If you use `gcc`, specify `CFLAGS="-ansi -fno-builtin"`.

Once `rcc` is built with the host C compiler, run the test suite to verify that `rcc` is working correctly. The commands in `src/makefile` run the shell script `src/run` on each C program in the test suite, `tst/*.c`. It uses the driver, `lcc`, so you must have the driver installed before testing `rcc`. The *target−system* combination is read from the variable `TARGET`, which is specified when invoking `make`:

```
% make -f ../../src/makefile TARGET=mips-ultrix test
../rcc -target=mips-ultrix 8q:
../rcc -target=mips-ultrix array:
../rcc -target=mips-ultrix cf:
../rcc -target=mips-ultrix cq:
../rcc -target=mips-ultrix cvt:
../rcc -target=mips-ultrix fields:
../rcc -target=mips-ultrix front:
../rcc -target=mips-ultrix incr:
../rcc -target=mips-ultrix init:
../rcc -target=mips-ultrix limits:
../rcc -target=mips-ultrix paranoia:
../rcc -target=mips-ultrix sort:
../rcc -target=mips-ultrix spill:
../rcc -target=mips-ultrix stdarg:
../rcc -target=mips-ultrix struct:
../rcc -target=mips-ultrix switch:
../rcc -target=mips-ultrix wf1:
../rcc -target=mips-ultrix yacc:
```

For each C program in the test suite, `src/run` compiles the program and uses `diff` to compare the generated assembly code with the expected code (the MIPS code expected for `tst/8q.c` is in `mips/ultrix/tst/8q.sbk`, etc.). If there are differences, the script executes the generated code with the input given in `tst` (the input for `tst/8q.c` is in `tst/8q.0`, etc.) and compares the output with the expected output (the expected output from `tst/8q.c` on the MIPS is in `mips/ultrix/tst/8q.1bk`, etc.). The script also compares the diagnostics from the compiler with the expected diagnostics.

On some systems, there may be a few differences between the generated code and the expected code. These differences occur because the expected code is generated by cross compilation on a MIPS and the least significant bits of some floating−point constants differ from those bits in constants generated on your system. There should be no differences in the output from executing the test programs.

The `../rcc` and `−target=mips−ultrix` preceding the name of each test program in the output above indicate the compiler and the target, e.g., "`../rcc` is generating code for a `mips` running the `ultrix` operating system."

Next, build `rcc` again using the just−built `rcc`:

```
%  make -f ../../src/makefile TARGET=mips-ultrix triple
rm -f *.o
make -f ../../src/makefile CC='lcc -B./ -d0.1 -A'
   CFLAGS='-Wf-target=mips-ultrix
   -I../../src/../include/`echo mips-ultrix|tr - /`
   -I../../src' LDFLAGS=''
lcc -B./ -d0.1 -A -c -Wf-target=mips-ultrix
   -I../../src/../include/`echo mips-ultrix|tr - /`
   -I../../src ../../src/alloc.c
...
lcc -B./ -d0.1 -A -c -Wf-target=mips-ultrix
   -I../../src/../include/`echo mips-ultrix|tr - /`
   -I../../src ../../src/x86.c
lcc -B./ -d0.1 -A -o rcc  alloc.o ... sparc.o x86.o
strip rcc
od rcc +8 >od2
rm -f *.o
make -f ../../src/makefile CC='lcc -B./ -d0.1 -A'
   CFLAGS='-Wf-target=mips-ultrix
   -I../../src/../include/`echo mips-ultrix|tr - /`
   -I../../src' LDFLAGS=''
lcc -B./ -d0.1 -A -c -Wf-target=mips-ultrix
   -I../../src/../include/`echo mips-ultrix|tr - /`
   -I../../src ../../src/alloc.c
...
lcc -B./ -d0.1 -A -c -Wf-target=mips-ultrix
   -I../../src/../include/`echo mips-ultrix|tr - /`
   -I../../src ../../src/x86.c
lcc -B./ -d0.1 -A -o rcc  alloc.o ... sparc.o x86.o
strip rcc
od rcc +8 >od3
cmp od[23] && rm od[23]
```

This command builds `rcc` twice; once using the `rcc` built by `cc` and again using the `rcc` built by `lcc`. After building each version, an octal dump of the resulting binary is made, and the two dumps are compared. They should be identical, as shown at the end of the output above. If they aren't, our compiler is generating bad code; contact us.

The final version of `rcc` should also pass the test suite; i.e., the output from

```
make -f ../../src/makefile TARGET=mips-ultrix test
```

should be identical to that from the previous `make test`.

Now install the final version of rcc:

```
% cp rcc /usr/local/lib/rcc
```

where the destination is the location chosen for rcc in Sec. 2.

On some systems, you may be able to use environment variables and make's -e option to avoid specifying TARGET on each make command, and the make commands described above can be done with a single command:

```
% setenv TARGET mips-ultrix
% cd mips/ultrix
% make -e -f ../../src/makefile test triple test clean
```

The command

```
make -f ../../src/makefile clean
```

cleans up, but does not remove rcc, and

```
make -f ../../src/makefile clobber
```

cleans up and removes rcc.

The code generators for the other targets can be tested by running make from the appropriate target-specific directory and setting some environment variables to control what src/run does. For example, if you built mips/ultrix/rcc and installed it in /usr/local/lib/rcc, you can test the SPARC code generator for the SunOS operating system as follows.

```
% setenv REMOTEHOST noexecute
% setenv BUILDDIR /usr/local/lib/
% cd sparc/sun
% make -f ../../src/makefile RCC= TARGET=sparc-sun test
/usr/local/lib/rcc -target=sparc-sun 8q:
/usr/local/lib/rcc -target=sparc-sun array:
/usr/local/lib/rcc -target=sparc-sun cf:
/usr/local/lib/rcc -target=sparc-sun cq:
/usr/local/lib/rcc -target=sparc-sun cvt:
/usr/local/lib/rcc -target=sparc-sun fields:
/usr/local/lib/rcc -target=sparc-sun front:
/usr/local/lib/rcc -target=sparc-sun incr:
/usr/local/lib/rcc -target=sparc-sun init:
/usr/local/lib/rcc -target=sparc-sun limits:
/usr/local/lib/rcc -target=sparc-sun paranoia:
/usr/local/lib/rcc -target=sparc-sun sort:
/usr/local/lib/rcc -target=sparc-sun spill:
/usr/local/lib/rcc -target=sparc-sun stdarg:
/usr/local/lib/rcc -target=sparc-sun struct:
/usr/local/lib/rcc -target=sparc-sun switch:
/usr/local/lib/rcc -target=sparc-sun wf1:
/usr/local/lib/rcc -target=sparc-sun yacc:
```

As above, src/run compares the SPARC code generated with what's expected. There should be no differences. Setting REMOTEHOST to noexecute suppresses the assembly and execution of the generated code. BUILDDIR gives the directory that holds rcc, and specifying RCC= to make insures that rcc is not rebuilt in the sparc/sun directory.

If you set REMOTEHOST to the name of a SPARC machine to which you can rlogin, src/run will rcp the generated code to that machine and execute it there, if necessary. See src/run for the details.

5.1 Using lcc as Cross–Compiler

Once everything is installed, you can use lcc as a cross compiler. The options -S and -Wf-target=*target-system* generate assembly code for the specified target, which is any of those listed in the file src/bind.c. For example,

```
% lcc -Wf-target=sparc-sun -S tst/8q.c
```

generates SPARC code for tst/8q.c in 8q.s.

lcc can also generate code for a "symbolic" target. This target is used routinely in front-end development, and its output is a printable representation of the input program, e.g., the dags constructed by the front end are printed, and other interface functions print their arguments. You can specify this target with the option -Wf-target=symbolic. For example,

```
% lcc -Wf-target=symbolic -S tst/8q.c
```

generates symbolic output for tst/8q.c in 8q.s. Finally, the option -Wf-target=null specifies the "null" target for which lcc emits nothing and thus only checks the syntax and semantics of its inputs files.

5.2 Installing the Code-Generator Generator

The directory lburg holds the source code for the code-generator generator. To build lburg, execute the commands

```
% cd lburg
% make
```

You must use lcc or another ANSI C compiler, e.g., gcc, because lburg is written in ANSI C.

To test lburg, use it process one of the machine descriptions in src/*.md and compare the output with the corresponding src/*.c file. For example,

```
% lburg <../src/x86.md | diff ../src/x86.nw -
40c40
< generated at Mon Dec  5 17:49:52 1994
---
> generated at Tue Dec  6 13:35:42 1994
```

processes src/x86.md and compares the output with what's expected. You may get 1-2 lines of differences, because lburg's output includes a timestamp and a version stamp. Move lburg to the desired installation directory. The command "make clean" cleans up, but does not remove lburg, and "make clobber" cleans up more and removes lburg.

src/makefile includes three rules for building src/mips.c, src/sparc.c, and src/x86.c from the corresponding machine descriptions in src/mips.md, src/sparc.md, and src/x86.md. These rules are commented out in the distributed src/makefile; remove the leading "#" to uncomment these lines, if you want to run lburg whenever these src/*.md files are changed. You may also want to remove the leading "#" on the rm command in the clobber rule, if you want "make clobber" to remove the src/*.c files generated by lburg.

Chapters 13-15 in *A Retargetable C Compiler: Design and Implementation* describe the overall structure of lcc's code generator, and the use of lburg. Chapters 16-18 describe the target-specific parts, including the lburg rules, for the MIPS, SPARC, and x86 architectures. The easiest way to write a code generator for a new target is to make a copy of the .md file for the architecture that's closest to your intended target, and edit the copy, using the corresponding chapter as a guide.

For example, the following steps add a code generator for the PowerPC under AIX.

1. Copy `src/mips.md` to, say, `src/power.md`, and edit this copy into a code generator for the PowerPC.

2. Edit `src/makefile`: Add `power.o` to the definition of `OBJS`, and add the following rules.

   ```
   power.o:        $(SRC)/power.c;        $(CC) -c $(CFLAGS) $(SRC)/power.c
   $(SRC)/power.c: $(SRC)/power.md;       lburg <$(SRC)/power.md >$(SRC)/power.c
   ```

3. Create the directories `power`, `power/aix`, and `power/aix/tst`.

4. Bind the new code generator to the front end by including it in the initialization of `bindings` in `src/bind.c` as described in Section 5.11 of *A Retargetable C Compiler*. For example, if the variable `powerIR` is the interface record for the PowerPC code generator, add the declaration

   ```
   extern Interface powerIR;
   ```

   to the beginning of `src/bind.c`, and add the line

   ```
   "power-aix",       &powerIR,
   ```

   to the initialization of `bindings`.

Once these steps are completed, the commands

```
% cd power/aix
% make -f ../../src/makefile TARGET=power-aix
```

build an `rcc` that includes the new code generator. It's a good idea to do all these steps changing `src/power.md` as little as possible to get the machinery in place first, then finish editing `src/power.md` into the new code generator.

6. lcc on a PC

This section describes installing and operating `lcc` under MS−DOS. Read all sections above first. The DOS distribution does not support `lcc`'s `-b`, `-g`, and `-p` options.

On the X86, `lcc` emits assembler code that uses a 32−bit flat address space. It requires an external assembler, linker, library, and DOS extender. We assemble `lcc`'s emitted code with Borland Turbo Assembler 4.0 or Microsoft's MASM 6.11, and we turn the resulting object code into a `.exe` file using the linker, library, and DOS extender from Borland C++ 4.02 plus Borland PowerPack for DOS. Warning: Borland's start−up code initializes the floating−point unit to convert floats and doubles to integers by rounding instead of truncating them, which the ANSI Standard specifies.

Small changes to `src\x86.c` have allowed it to work with the linker, library, and DOS extender from Symantec C++ 6.1.

6.1 Installing lcc under DOS

The distribution includes the following files:

`x86\dos\cpp.exe`
  the preprocessor
`x86\dos\rcc.exe`
  the compiler proper

`x86\dos\lcc.exe`
: the compiler's driver

`x86\dos\32rtm.exe`
: Borland PowerPack runtime manager

`x86\dos\dpmi32vm.ovl`
: Borland PowerPack DPMI server

`x86\dos\windpmi.386`
: Borland PowerPack Windows DPMI driver

Copy the `.exe` and `.ovl` files to a directory in your `PATH`. If you want to use `lcc` in a full−screen DOS window under Windows 3.1, copy `windpmi.386` to your `\windows\system` directory, and add the lines

```
[386Enh]
device=windpmi.386
```

to your `system.ini` (which is located in `\windows`).

`lcc.exe` is a prebuilt driver for use with Borland's Turbo Assembler 4.0, Borland C++ 4.02, and Borland PowerPack for DOS. To use some other combination of assembler, linker, library, and DOS extender, or to change the path names compiled into the driver, you will have to build a new driver. You'll probably have to modify the driver and `include\x86\dos\*.h` if you want to use them. The source code for the PC version of the driver is in `etc\lcc-pc.c` and `etc\bc4.c`. It is unlikely that you'll need to rebuild `cpp.exe` or `rcc.exe`.

`lcc` comes with a preprocessor and ANSI include files (`include\x86\dos\*.h`) that work with the library and DOS extender included with Symantec C++ 6.1. Include files are generally tailored, however, for a specific library, so use the headers that come with your library, or adapt `lcc`'s headers to work with your library.

If you use headers that exploit language extensions, you must also use their companion preprocessor, or you must protect `cpp.exe` from the extensions, because `cpp.exe` obeys the ANSI specification. For example, when we use `lcc` with the library from Borland C++ 4.02, we use Borland's headers too, and they annotate many declarations with a non−ANSI attribute, `__cdecl`. The distribution's `cpp.exe` won't recognize `__cdecl`, so we use the `cpp.exe` option `-D__cdecl=` to eliminate this extension, or we use Borland's preprocessor.

6.2 Using lcc under DOS

The default executable file is `a.exe`, and object files have the extension `.obj`. The other extensions are as described in Sec. 4, above, including assembler files, which end in `.s`, not `.asm`.

The Unix driver for `lcc` invokes the process for each compiler subphase that's needed. In principle, the DOS driver could do likewise, but doing so fails under at least some DOS extenders. For maximum portability, the distributed DOS driver thus merely emits a command file, into `dolcc.bat` in the current directory, which must be run to compile anything; for example,

```
C:> lcc 8q.c
C:> dolcc.bat
```

compiles, assembles, and links `8q.c`. and leaves the executable in `a.exe`. `lcc.exe` writes commands to `dolcc.bat` that generate and consume temporary files. These files have names like `c:\tmp\lcc`*ddddd*`.*`, so create the directory `c:\tmp`, if it doesn't exist.

Users running programs like doskey can easily create a macro that abbreviates this idiom, but makefiles and batch files must invoke dolcc.bat explicitly. If the library routines system or execv work with your DOS extender, you might be able to modify the routine execute in etc\lcc-pc.c to avoid this irritation.

Note that the distributed lcc.exe obviates the option -v -v decsribed in Sec. 4 because it never executes commands anyway. Also, it complicates the idiom that saves the preprocessed source in a file. That is,

```
C:> lcc -E 8q.c >8q.i
```

fails to capture the preprocessed 8q.c in 8q.i because lcc.exe creates but does not execute a preprocessor command.

6.3 Building lcc under DOS

It is not likely that you'll need to rebuild cpp.exe, nor is it likely that you'll need to rebuild rcc.exe, unless your assembler or library, say, requires the compiler to emit different assembler code. Read the rest of this section only if you need to modify rcc.exe.

On MS-DOS systems, several batch files replace the Unix make and shell files described in the sections above. src\triple.bat replaces the make triple process described in Sec. 5. It calls src\lccn.bat to compile lcc's modules using lcc.exe. src\test.bat runs lcc's test suite. It calls src\run.bat once for each test program. The batch files compile the test files in the directory x86\dos\tst, but they compile the compiler itself in the src directory. lccn.bat and run.bat include dolcc.bat as a hard path name; edit them if you changed the driver to deliver its commands elsewhere.

src\triple.bat builds rcc.exe three times. It builds the first with Borland C++ 4.02, and builds the other two with the rcc.exe built just before. For the first of these three stages, triple.bat copies the generated executable to directory src\rcc1. The second and third stages do likewise with directories src\rcc2 and src\rcc3, and they move the .s files there as well, so that they can be compared. Clear these directories once all tests pass.

Each stage runs src\test.bat. Once all tests pass, go to the directory x86\dos\tst and run clean.bat, which cleans up.

Ideally, rcc2\rcc.exe and rcc3\rcc.exe or rcc2*.obj and rcc3*.obj would be identical, but at least some DOS assemblers and linkers emit slightly different object and executable files when rerun on some fixed assembler modules, so triple.bat can confirm that lcc consistently emits the same code for itself only by comparing rcc2*.s with rcc3*.s.

7. Reporting Bugs

lcc is a large, complex program. We find and repair errors routinely. If you think that you've found a error, follow the steps below, which are adapted from the instructions in Chapter 1 of *A Retargetable C Compiler: Design and Implementation*.

1. If you don't have a source file that displays the error, create one. Most errors 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.

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*, 2nd edition by B. W. Kernighan and D. M. Ritchie (Prentice Hall, 1988), or the appropriate section in *C: A Reference Manual*, 3rd edition by S. B. Harbison and G. L. Steele, Jr. (Prentice Hall, 1991). If `lcc` silently generates incorrect code for some construct, please include the corrupt assembly 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` from `ftp.cs.princeton.edu` in `pub/lcc`. A `README` file there gives acquistion details, and a `LOG` file reports what errors were fixed and when they were fixed. If you report a error that's been fixed, you might get a canned reply.

7. Send your program by electronic mail 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.

8. Keeping in Touch

There is an `lcc` mailing list for general information about `lcc`. To be added to the list, send a message with the 1–line body

```
subscribe lcc
```

to `majordomo@cs.princeton.edu`. This line must appear in the message body; "Subject:" lines are ignored. To learn more about mailing lists served by `majordomo`, send a message with the 1–word body "`help`" to `majordomo@cs.princeton.edu`. Mail sent to `lcc@cs.princeton.edu` is forwarded to everyone on the mailing list.

There is also an `lcc-bugs` mailing list for reporting bugs; subscribe to it by sending a message with the 1–line body

```
subscribe lcc-bugs
```

to `majordomo@cs.princeton.edu`. Mail addressed to *lcc–bugs@cs.princeton.edu* is forwarded to everyone on this list.

Chris Fraser / cwf@research.att.com
David Hanson / drh@cs.princeton.edu
$Revision: 1.14 $ $Date: 1994/12/16 15:31:42 $