A Tool for Automated Optimization Correctness Testing

Rob Shillingsburg

October 17, 1993

[0]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

Chapter 1

Introduction

As the Massively Scalar Compiler grows in size and complexity, we desire an automated method of running correctness tests. Ideally, we would like to have a large suite of Fortran test programs which can be compiled (using our compiler) and run automatically, with the output of the programs compared against known correct results. We want the user to be able to specify which programs from the test suite should be run. We also want to be able to specify an arbitrary sequence of transformations for each test, and we want multiple users to be able to run tests concurrently. Finally, since the test runs could conceivably take a very long time, we'd like to have a human-readable summary file produced for each test run.

1.1 Components

Our testing tool is made up of the following pieces:

- A suite of test program sources, written in Fortran or ILOC and arranged at the leaves of a tree of directories rooted at `/compiler/test`. Each leaf directory must contain exactly one test program. In addition to this central test suite, individual users may wish to accumulate their own private test suites.
- A collection of files each named `Results`, one for each program in the test suite. Each `Results` file contains the correct (canonical) output of a test suite program. The `Results` files are stored in the appropriate leaf directories of the test hierarchy.
- An executable program called `ctest`, which the user executes to begin a testing run. `ctest` starts up `gnumake` processes to do the work of compiling and running the tests.
- A single GNU makefile called `duplicate.script`, which tells `gnumake` how to construct a copy of the test hierarchy.
- A single GNU makefile called `test.script`, which controls the details of compiling and running test programs.
- Copies of a GNU makefile called `Ctest.script`, placed at each leaf directory in the test hierarchy and customized for each test program. These files tell a `gnumake` process what inputs and outputs each test program has.

The `nuweb` program which produced this document also produces `ctest.c`, `duplicate.script`, `test.script` and `Ctest.script`.

1.2 Functionality

To begin a test run, the user invokes the program `ctest`, supplying on the command line the parameters of the test, including the following:

transformations The names and ordering of the compiler transformations to be tested.

test directory The name of the root of a directory tree containing the test programs to be used.

user directory The name of a directory in which to compile the test programs and to store the log files from the test.

The command-line syntax and the other available options are are discussed in Section 2.1.

Our testing tool functions by making a copy of the test hierarchy, that is, the directory subtree rooted at the test directory. The testing facility creates this directory tree (known as the *duplicate hierarchy*) in the user directory. The test mechanism then compiles (using the given transformations) and runs each of the test programs in the duplicate hierarchy, comparing the output to the canonical output stored in the `Results` files. Tests whose output does not match `Results` are considered unsuccessful.

Once testing is complete, the test mechanism deletes the duplicate directories which contain only successful tests, leaving the unsuccessful test programs and their output intact for later inspection. However, the duplicate of the original test directory is always left intact, because that duplicate directory contains the log files for the test run. We discuss these files in Section 1.2.3.

1.2.1 Creating the duplicate hierarchy

After parsing the user's command-line input, `ctest` creates a directory (the *duplicate directory*) inside the user directory. The duplicate directory has a name of the form *dir*`.`*date*, where *dir* is the name of the test directory, and *date* is the current date in the form *YY.MM.DD.hh.mm.ss*. The duplicate directory will hold the log files as well as serve as the root of the duplicate hierarchy.

The `ctest` program then spawns a `gnumake` process in the test directory to create the rest of the duplicate hierarchy. This `gnumake` process uses the file `duplicate.script` as its makefile. The directives in `duplicate.script` first check to see if the test directory is a leaf, then perform further actions based on the outcome of the check.

- If the test directory is not a leaf, the `gnumake` process creates duplicates of the subdirectories of the test directory. It creates the duplicates in the duplicate directory. It then starts up a recursive `gnumake` process in each of the duplicated directories.

- If the test directory is a leaf, the `gnumake` process creates symbolic links in the duplicate directory which point back to each of the files in the test directory.

When the recursive `gnumake` algorithm is finished, the duplicate hierarchy will be complete, and each leaf of the duplicate hierarchy will contain links back to the files in the corresponding directories in the test hierarchy.

1.2.2 Compiling and running test programs

When the duplicate hierarchy is complete, `ctest` starts up another `gnumake` process, this time using the file `test.script` as its makefile. The new `gnumake` is created in the duplicate of the test directory. It also performs different actions based on whether the duplicate directory is a leaf.

If the duplicate directory is not a leaf, the `gnumake` spawns recursive `gnumakes` in each of the subdirectories it finds. When the recursive processes finish, the parent `gnumake` attempts to delete the subdirectories. If the test programs in a particular subdirectory were successful, the child `gnumake` running in that subdirectory would have emptied the subdirectory, allowing the deletion of the subdirectory by the parent to be successful. Otherwise, the deletion will not work, and the subdirectory will remain. This strategy eventually removes all duplicate directories which contain only successful tests.

Compiling

If the duplicate directory is a leaf directory, then it must contain source files. The `test.script` makefile contains directives to compile and run two types of programs: those written in Fortran only and those written in both Fortran and ILOC. These directives inspect the files present in the duplicate directory (remember that the "files" are symbolic links back to the files in the test directory), and compile the program in the duplicate directory using the appropriate compilers and translators. These are:

`f2i` converts a Fortran file to an ILOC file (using `a2i`, the AST to ILOC converter).

Transformations specified by the user translate ILOC to ILOC, performing various optimizations and transformations. Examples of these are `dead`, `cprop` and `address`.

`i2c` converts an ILOC file into a C file.

`xlc` compiles a C file.

`xlf` compiles a "driver" or `.io.f` Fortran file.

There are, of course, two ways to deal with a Fortran file: `f2i` and `xlf`. We use `f2i` only when there are no ILOC source files in the test directory; that is, we build all the ILOC files using `f2i`. We use `xlf` when there are both Fortran and ILOC sources present; in this case we consider the Fortran files to be "driver" routines which should be compiled using `xlf`. Note also that `f2i` produces not only ILOC output, but files ending in `.io.f` as well. These are Fortran files which implement I/O procedures not directly supported by ILOC. We compile `.io.f` files using `xlf`.

When all the files have been compiled into object form, we use `xlf` to link the objects into an executable file named `main`.

Running

We then run the program using the input specified in the `Ctest.script` file (if any). When execution is finished, we use `diff` to compare the output of the program (whose filename(s) are specified in `Ctest.script`) to the `Results` file for that test program.

If a test program's output matches `Results`, then the test for that program is considered successful, and `gnumake` removes all files from the duplicate directory. However, if a test program's output does not match `Results`, that program and its associated files are left intact in the directory and a message is issued to the log file and the screen.

1.2.3 Log Files

The test mechanism produces two log files for each test run; both are created in the directory named *dir.date* that the `ctest` program makes. One file is called `MAKE.LOG`; it contains the largely unreadable transcript of all of the commands issued by the `gnumake` processes, as well as all the complaints and messages sent to standard error by those commands.

The other, more interesting file is called `TEST.LOG`, and it contains human-readable status messages presented by the test mechanism to mark the notable events of a test run. `TEST.LOG` contains the following messages:

- `Begin ctest run in hierarchy` *dir*. This message is issued once at the beginning of the test run.
- `In directory` *dir*. This message is issued when the test mechanism enters *dir*.
- `ERROR:` *dir*: *errormsg*. This message means that an error has occurred during the processing of *dir*.
- `Finished directory` *dir*. This message is issued when the test mechanism completes all processing of *dir*.
- `End ctest run.` This message is issued at the end of the test run.

Chapter 2

Using, maintaining and modifying the test facility

This chapter is devoted to the human interface of the testing mechanism. The first section is a guide to the command-line options of the the testing mechanism. The second section gives information for the maintainer of the test suite, including instructions for setting up and maintaining the test suite's directory structure and test programs. The third section provides tips on adding functionality to the testing facility.

2.1 Command-line options

The command-line interface of `ctest` looks like this:

`ctest` *testDir* `[-D` *userDir*`] [-KMQRacdglnpv{`*cmd*`}] [-C` *i2c*`] [-F` *f2iOpt*`] [-I` *i2cOpt*`]`

We have a convention for naming `ctest`'s command-line flags: we use lowercase letters for transformations to be applied to test programs, and we use capital letters for all other directives. The items on the command line may be given in any order. The following items may be supplied:

- The name of a directory, which `ctest` will use as the test directory. There is no default, so `ctest` will abort if the user does not specify a test directory.
- The `-D` flag, followed by the name of a directory, which `ctest` will use as the user directory (*i.e.,* the directory in which to perform compilation and testing). The default is the current directory.
- Flags which indicate transformations to be performed on test programs. By default, no transformations are performed. Order and repetitions are significant, and the flags may be combined, as in `-dvd`. Each transformation is taken to be the version which appears first in the user's path; the `-{`*cmd*`}` option provides a way to specify other transformations. Each of the transformation flags may be immediately followed by a list of flags to be passed to the transformation, enclosed by square brackets. Remember to enclose the square brackets and curly braces in quotes to protect them from the shell (*e.g.,* `ctest "-l[-c]pvcd"`). These are the flags and their meanings:

  `-a` the assertion inserter, `assert`.

  `-c` the constant propagator, `cprop`.

  `-d` the dead code eliminator, `dead`.

  `-g` the global value numberer, `gval`.

  `-l` the partial-redundancy eliminator, `partial`.

  `-n` the excess-control-flow remover, `clean`.

`-p` the loop peeler, `peel`.

`-r` `reassociate`, which rearranges expressions to expose loop-invariant expressions.

`-v` the local value numberer, `valnum`.

`-{`*cmd*`}` This option issues the transformation *cmd*. The given command must be an ILOC filter; in addition, it must be either an absolute pathname or a pathname relative to the current directory. That is, the environment's path is not searched.

- The `-C` flag, followed by the name of a program to use in place of `i2c`. Absolute or relative pathnames are allowed, but no "path chasing" will be performed.
- The `-F` flag, followed by a string to be included on the command line of `f2i`. If the string contains whitespace, it should be enclosed in quotes.
- The `-I` flag, followed by a string to be included on the command line of `i2c`. If the string contains whitespace, it should be enclosed in quotes.
- The `-M` (make only) flag, which tells the testing mechanism to compile the test programs, but not to run any tests or delete any files.
- The `-K` (keep files) flag, which tells the testing mechanism to compile and run the test programs, but not to delete any files. `-M` takes precedence over `-K`.
- The `-R` flag, which directs the testing tool to rebuild the `Results` file for each program in the test hierarchy. `-R` supercedes `-M`. The `-R` flag should be used only by the maintainer of the test suite, because `-R` modifies the contents of the test hierarchy. The uses of this flag are more fully described in Section 2.2.3.
- The `-Q` flag, which makes `ctest` run in quiet mode. Normally, `ctest` prints status messages to `/dev/tty`, to keep the user appraised of the test run's progess. However, there are certain situations where these messages are not desirable (*e.g.,* when using `ctest` with the `at` command). Quiet mode turns off these status messages.

Following are some examples of using the test mechanism:

`ctest /compiler/test/spec -D ~/test-results -d`

This command runs tests of the dead code eliminator (`-d`) on all programs found in the directory `/compiler/test/spec`, doing all compilation (and storing of log files) in the directory `~/test-results`. After testing, the test mechanism deletes all successful test programs from `~/test-results`.

`ctest /compiler/test/fmm -M`

This command compiles the programs found in `/compiler/test/fmm`; the compilation occurs in the current directory. No test programs are run, and no files are deleted after processing is complete; the programs are simply compiled for browsing and inspection by the user.

`ctest /compiler/test/fmm/fmin "-l[-c]cdn" -K`

This command compiles the program found in `/compiler/test/fmm/fmin`, using the transformations `partial -c`, `cprop`, `dead`, and `clean`. Compilation occurs in the current directory. The test program is run, but the `-K` flag tells the test mechanism not to delete the test program after comparing the output to the `Results` file.

`ctest /compiler/test/fmm "-lv{./dead}[-ddd]n"`

This command compiles and runs the programs found in `/compiler/test/fmm`, using the transformations `partial`, `valnum`, `./dead -ddd`, and `clean`.

`ctest /compiler/test -C ./i2c -I -n`

This command compiles and runs the programs found in `/compiler/test`. Instead of using the installed version of `i2c`, this test run will use `./i2c`, with the `-n` option.

2.2 Test suite maintainer's guide

2.2.1 The test hierarchy

As was stated in the introduction, the central test hierarchy is rooted at `/compiler/test`. This hierarchy need not be world- or group-writable, because performing test runs does not modify the test suite. The only user who needs to have write access to the test hierarchy is the test suite maintainer, who is the person responsible for installing and maintaining test programs. In our group, the test suite maintainer's account name is `tester`. Individual users who desire to have certain test programs installed in the test suite should contact the test suite maintainer. Individual users may also create and maintain their own test suites, possibly in directories named `~/test`.

As mentioned previously, a test hierarchy contains a set of leaf directories and a (possibly empty) set of non-leaf directories. The source files of the test programs reside at the leaves, while the interior directories serve to organize the test programs into sensible groups. The following rules apply for leaf directories in a test hierarchy:

1. Every leaf directory must contain the source files for exactly one test program.
2. Each test program must be written in either Fortran or a combination of Fortran and ILOC. If two source files `foo.f` and `foo.i` exist, only `foo.i` is used to make `foo.o`; `foo.f` is ignored.
3. Each leaf directory must contain a properly-modified `Ctest.script` file (see Section 2.2.2).
4. The filenames `main`, `MAKE.LOG`, `TEST.LOG`, `DIFF`, `ERROR`, `Results` and `Newresults` are reserved for use by the testing mechanism, so these filenames should not be used for the source or output files of any test program.

No rules apply to non-leaf directories. However, it is important to note that all stray files in non-leaf directories (including test programs) will be ignored by the testing mechanism.

2.2.2 `Ctest.script` files

Each leaf directory in a test hierarchy must contain a `Ctest.script` file. This file gives information to the testing mechanism concerning the input and output requirements of the test program which it accompanies. Therefore, the model `Ctest.script` file which we present here must be customized for each test program.

"Ctest.script" 7 ≡
```
INPUT =
OUTPUT =
```
◇

`Ctest.script` contains two lines, each of which defines a `gnumake` variable. The test suite maintainer should set the `INPUT` variable to a Bourne shell (`sh`) fragment which causes the proper input to be sent to the test program; this fragment will be included on the command line which runs the compiled test program. For example, if the test program reads from the standard input, and the proper input is stored in the file `test_input`, the proper `Ctest.script` line would be:

```
INPUT = < test_input
```

If the test program requires its input file to be stated after, say, the `-f` option, the proper `Ctest.script` line would be:

```
INPUT = -f test_input.
```

If the test program has no input requirements, `INPUT` should be left undefined.

The `OUTPUT` variable should be set to the file names of the output files produced by the test program, if any. The standard output and standard error are automatically captured by the testing mechanism, and need not be specified in the `OUTPUT` variable.

2.2.3 Building `Results` files

Once all of the source files, input files and `Ctest.script` files are in place in the test hierarchy, the only step left is to create the `Results` files. Each test program has a `Results` file associated with it which contains the correct output of that test program.

The testing mechanism provides a way to automatically create the `Results` files. The `-R` command-line option to `ctest` tells the testing mechanism to compile and run test programs as it normally would. However, instead of comparing the output to existing `Results` files, the testing mechanism just copies the new output into the test hierarchy, giving it the name `Results`.

2.3 Adding functionality to the test facility

The test facility is designed with modification in mind. It utilizes a two-pass structure, where one pass builds the duplicate hierarchy and another pass does something with it. This second pass can be replaced, thereby swapping in a different functionality but retaining the overall architecture.

In practice, the best way to make substantial changes to the test facility is to copy its source `nuweb` file and modify the copy, rather than trying to squeeze extra functionality into the existing tools and letting the user choose which procedures to use. The copy-then-redesign approach helps keep down command-line clutter and aims to keep each facility simple. The syntax `gnumake` uses is complex enough for simple things; it could become utterly unbearable for large, multifunction test facilities.

Chapter 3

Processing user input

The user begins a test run by executing the `ctest` program. The `ctest` program looks like this:

"ctest.c" 9a ≡
  ⟨Include files 9b, ... ⟩
  ⟨Global variables 10a, ... ⟩
  ⟨Prototypes 33f⟩
  ⟨Functions 14b⟩
  ⟨The `main` routine 9c⟩
  ◇

We need to include the file `Compiler_Types.h`, as it contains the definitions of portable data types which are to be used in all Massively Scalar Compiler Project components. In addition, we will be doing I/O and some string processing, so we include the appropriate header files.

⟨Include files 9b⟩ ≡

```
#include <Compiler_Types.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
```

◇

Macro defined by scraps 9b, 10b, 18b.
Macro referenced in scrap 9a.

The `ctest` program has three major responsibilities: parsing user input, creating the duplicate hierarchy, and compiling and running the test programs. The `main` routine performs these tasks in sequence.

⟨The `main` routine 9c⟩ ≡

```
Void main(Unsigned_Int argCount, Char **argVector)
{
  ⟨Parse the command line 11a⟩
  ⟨Create the duplicate hierarchy 17b⟩
  ⟨Perform tests 23⟩
  exit(0);
}◇
```

Macro referenced in scrap 9a.

The remainder of this chapter focuses on the first of `ctest`'s tasks, processing the user's command-line input.

3.1 Storage for command-line data

Before inspecting the command line, we need to declare storage and initial values for the information we expect to uncover. Here we declare character arrays to hold the names of the test directory and user directory, called testDir and userDir. The string i2c holds the name of the user-supplied version of i2c. The strings i2cFlags and f2iFlags will hold the flags the user wishes to pass to i2c and f2i.

We also declare an character array to hold the list of transformations specified by the user. The array, transformations, has the format "| opt_1 | ... | opt_n", where opt_i is the name of any ILOC transformation program, such as dead, cprop or partial. Repetitions are both allowed and significant, and the initial value is the empty string.

⟨Global variables 10a⟩ ≡
```
static Char testDir[MAXPATHLEN] = "";
static Char userDir[MAXPATHLEN] = "";
static Char i2c[MAXPATHLEN] = "";
static Char i2cFlags[200] = "";
static Char f2iFlags[200] = "";
static Char transformations[500] = "";
```
◇

Macro defined by scraps 10ac, 17a, 21b.
Macro referenced in scrap 9a.

The MAXPATHLEN array dimension is defined in the include file sys/param.h. It is equal to the longest possible path name.

⟨Include files 10b⟩ ≡
```
#include <sys/param.h>
```
◇

Macro defined by scraps 9b, 10b, 18b.
Macro referenced in scrap 9a.

Some of the user's input takes on essentially Boolean values. However, since we want to pass ths input to gnumake in the form of a gnumake variable, we will represent them with pointers to character strings. Each of these variables will point to either an empty string or the string "TRUE", for false and true values, repectively. The variables which we will treat this way are results, noDelete, noRun and quiet. These variables have the following meanings:

- A true results value tells the testing mechanism to make the Results files in the test hierarchy; results is set by -R.
- A true noDelete value tells the testing mechanism not to delete the compiled test programs after running them; noDelete is set by -M and -K.
- A true noRun value tells the testing mechanism not to run the test programs, but only to compile them; noRun is set by -M.
- A true quietMode value tells the testing mechanism not to print status messages to /dev/tty.

⟨Global variables 10c⟩ ≡
```
static Char FALSE_STRING[] = "";
static Char TRUE_STRING[] = "TRUE";

static Char *results = FALSE_STRING;
static Char *noDelete = FALSE_STRING;
static Char *noRun = FALSE_STRING;
static Char *quietMode = FALSE_STRING;
```
◇

Macro defined by scraps 10ac, 17a, 21b.
Macro referenced in scrap 9a.

3.2 Parsing

We parse the command line for two types of items: switches beginning with `-`, and the name of the test directory. We parse command-line switches using a `switch` statement. Inside the `switch`, `currChar` points to the next unread character in the current `argVector` entry. When we're done with the command line, we install defaults for unspecified values.

⟨Parse the command line 11a⟩ ≡
```
{
  Unsigned_Int argNumber;

  for (argNumber = 1; argNumber < argCount; argNumber++) {
    if (argVector[argNumber][0] != '-') {
      ⟨Parse the name of the test directory 11b⟩
    } else {
      Char *currChar = &argVector[argNumber][1];

      while (*currChar) {
        switch (*currChar++) {
          ⟨Parse a command-line switch 12a, ... ⟩
        }
      }
    }
  }
  ⟨Postprocess the user's input 15a⟩
}◇
```
Macro referenced in scrap 9c.

We save the name of the test directory by copying it into `testDir`. If a test directory has already been specified, we print an error message and terminate.

⟨Parse the name of the test directory 11b⟩ ≡
```
if (!testDir[0]) {
  strcpy(testDir, argVector[argNumber]);
} else {
  fprintf(stderr, "ctest: You may specify only one test directory.\n");
  ABORT;
}◇
```
Macro referenced in scrap 11a.

In the event that we find an unknown command-line switch, we print the following rather verbose error message, then quit.

⟨Parse a command-line switch 12a⟩ ≡
```
default:
  fprintf(stderr, "Usage: ctest <test_dir> [options] [transformations]\n");
  fprintf(stderr, "\t<test_dir>\ttest directory\n");
  fprintf(stderr, "    Options:\n");
  fprintf(stderr, "\t-D <user_dir>\tuser directory (default .)\n");
  fprintf(stderr, "\t-C <i2c>\tuser-specified i2c program\n");
  fprintf(stderr, "\t-I <i2c_flags>\tcommand-line options for i2c\n");
  fprintf(stderr, "\t-F <f2i_flags>\tcommand-line options for f2i\n");
  fprintf(stderr, "\t-M\t\tmake only; do not run tests\n");
  fprintf(stderr, "\t-K\t\tkeep files; do not delete tests after running\n");
  fprintf(stderr, "\t-R\t\tbuild Results files\n");
  fprintf(stderr, "\t-Q\t\tquiet mode; don't print messages to /dev/tty\n");
  fprintf(stderr, "    Transformations (pass arguments in brackets):\n");
  fprintf(stderr, "\t-a\t\tassert\n");
  fprintf(stderr, "\t-c\t\tcprop\n");
  fprintf(stderr, "\t-d\t\tdead\n");
  fprintf(stderr, "\t-g\t\tgval\n");
  fprintf(stderr, "\t-l\t\tpartial\n");
  fprintf(stderr, "\t-n\t\tclean\n");
  fprintf(stderr, "\t-p\t\tpeel\n");
  fprintf(stderr, "\t-r\t\treassociate\n");
  fprintf(stderr, "\t-v\t\tvalnum\n");
  fprintf(stderr, "\t-{cmd}\t\tuser-specified transformation\n");
  fprintf(stderr, "Don't forget to enclose brackets and braces in quotes!!\n");
  ABORT;
  break;
◇
```
Macro defined by scraps 12abc, 13a.
Macro referenced in scrap 11a.

If we see any of the following switches, we simply copy the next `argVector` entry into the appropriate character array.

⟨Parse a command-line switch 12b⟩ ≡
```
case 'D' :  strcpy(userDir, argVector[++argNumber]); break;
case 'C' :  strcpy(i2c, argVector[++argNumber]); break;
case 'F' :  strcpy(f2iFlags, argVector[++argNumber]); break;
case 'I' :  strcpy(i2cFlags, argVector[++argNumber]); break;
◇
```
Macro defined by scraps 12abc, 13a.
Macro referenced in scrap 11a.

For `-K`, we set `noDelete` to a true value; for `-M`, we set both `noDelete` and `noRun`. For `-R`, we set `results`, and for `-Q` we set `quietMode`.

⟨Parse a command-line switch 12c⟩ ≡
```
case 'K' :  noDelete = TRUE_STRING; break;
case 'M' :  noRun = noDelete = TRUE_STRING; break;
case 'R' :  results = TRUE_STRING; break;
case 'Q' :  quietMode = TRUE_STRING; break;
◇
```
Macro defined by scraps 12abc, 13a.
Macro referenced in scrap 11a.

The remaining flags represent ILOC-to-ILOC transformations. Each time we see one we will accumulate some characters into the `transformations` string. We use the `AddTransformation` function to update the string properly.

`AddTransformation` takes two arguments: a string naming a transformation, and a pointer to the current location in the current `argVector` entry. `AddTransformation` inserts the transformation name into the `transformations` string, followed by whatever characters it finds enclosed in square brackets in the `argVector` entry. `AddTransformation` returns the position in the `argVector` entry up to which it read.

⟨Parse a command-line switch 13a⟩ ≡

```
case 'a' :  currChar = AddTransformation("assert", currChar);  break;
case 'c' :  currChar = AddTransformation("cprop", currChar);  break;
case 'd' :  currChar = AddTransformation("dead", currChar);  break;
case 'g' :  currChar = AddTransformation("gval", currChar);  break;
case 'l' :  currChar = AddTransformation("partial", currChar);  break;
case 'n' :  currChar = AddTransformation("clean", currChar);  break;
case 'p' :  currChar = AddTransformation("peel", currChar);  break;
case 'r' :  currChar = AddTransformation("reassociate", currChar);  break;
case 'v' :  currChar = AddTransformation("valnum", currChar);  break;
case '{' :  ⟨Parse a curly-braced command 13b⟩ break;
◇
```

Macro defined by scraps 12abc, 13a.
Macro referenced in scrap 11a.

We also use `AddTransformation` on commands enclosed in curly braces; however, we have to process the command a little first. Here we count the number of characters in a command, issuing an error message if the count is zero or if there is no terminating curly brace.

⟨Parse a curly-braced command 13b⟩ ≡

```
{
  Unsigned_Int i;

  for (i = 0; currChar[i] != '}'; i++) if (!currChar[i]) {
    fprintf(stderr, "ctest: You must match '{' with '}'.\n");
    ABORT;
  }
  if (!i) {
    fprintf(stderr, "You must specify a command between '{' and '}'.\n");
    ABORT;
  }
  ⟨Process the curly-braced command 14a⟩
}◇
```

Macro referenced in scrap 13a.

When we've finished counting, we write a `NULL` terminator over the closing curly brace. Then we look at the first non-whitespace character of the command. If it is a slash, then the command is the full pathname of an executable program, and we pass the command and next unread `argVector` character to `AddTransformations`.

If the first non-whitespace character of the command is not a slash, then the command is a relative pathname of an executable program, and we need to prepend the command with the name of the current directory. We do this by finding the name of the current directory with `getwd`, then concatenating a slash and the command to the directory name. We then pass result of the concatenation to `AddTransformation`.

In both cases, the second argument we pass to `AddTransformation` is a pointer to the first unread character in the current `argVector` string. We pass `currChar + i + 1`.

⟨Process the curly-braced command 14a⟩ ≡
```
currChar[i] = '\0';
if (currChar[strspn(currChar, " \t")] == '/') {
  currChar = AddTransformation(currChar, currChar + i + 1);
} else {
  Char command[MAXPATHLEN];

  getwd(command);
  strcat(strcat(command, "/"), currChar);
  currChar = AddTransformation(command, currChar + i + 1);
}
◇
```
Macro referenced in scrap 13b.

The `AddTransformation` function takes two arguments: `name`, which is the name of the transformation, and `currChar`, which should be the value the `currChar` variable of the previous scope.

`AddTransformation` accumulates `name`'s characters into `transformations`, prepended with an or-bar. It then looks at the characters pointed to by `currChar`. If there is an opening square bracket at `currChar`, it also copies int `transformations` whatever it finds between the bracket and its corresponding closing bracket.

As `AddTransformations` puts characters into `transformations`, it needs to keep track of the location into which to write the next character. We keep a static variable called `transEnd` for this purpose.

`AddTransformations` returns a pointer to the character following the last character read from `currChar`. It it intended that the outer scope's `currChar` will be assigned this value.

> **WARNING**. The return value of `sprintf` is the number of characters copied, not counting the terminating NULL. However, although this is true for the RS/6000, it is not true for the Sun 4. Be careful when porting.

⟨Functions 14b⟩ ≡
```
Char *AddTransformation(Char *name, Char *currChar)
{
  static char *transEnd = transformations;

  transEnd += sprintf(transEnd, "| %s ", name);
  if (*currChar == '[') {
    while (*++currChar != ']') {
      if (*currChar) {
        *transEnd++ = *currChar;
      } else {
        fprintf(stderr, "ctest: You must match '[' with ']'.\n");
        ABORT;
      }
    }
    *transEnd++ = ' ';
    currChar++;
  }
  return currChar;
}◇
```
Macro referenced in scrap 9a.

3.3 Postprocessing

Having recorded the user's input in a meaningful form, we now want to make sure that certain variables have the proper default values and are in the proper form. In particular, we need process `userDir`, `testDir`, `i2c` and `noRun`.

However, the first step is to find the full pathname of the current directory. We use the C library function `getwd` perform this task.

⟨Postprocess the user's input 15a⟩ ≡

```
{
  Char currentDir[MAXPATHLEN];

  if (!getwd(currentDir)) fprintf(stderr, "ctest: %s\n", currentDir);
  ⟨Find full pathname of the user directory 15b⟩
  ⟨Find full pathname of the test directory 15c⟩
  ⟨Find full pathname of i2c 15d⟩
  ⟨Make results and noRun consistent 16⟩
}◇
```

Macro referenced in scrap 11a.

Next, we want to make sure that we have the full pathname of the user directory. If the user specified a user directory, we go to it (using `chdir`) and find its full pathname with `getwd`. If the user specified no user directory, we use `currentDir` for the user directory.

⟨Find full pathname of the user directory 15b⟩ ≡

```
if (userDir[0]) {
  if (chdir(userDir) == -1) {
    perror("ctest: user directory"); ABORT;
  }
  getwd(userDir);
  chdir(currentDir);
} else {
  strcpy(userDir, currentDir);
}◇
```

Macro referenced in scrap 15a.

We also want to make sure that we have the full pathname of the test directory. If the user specified a test directory, we go to it (using `chdir`) and find its full pathname with `getwd`. If the user specified no test directory, we print an error message and abort.

⟨Find full pathname of the test directory 15c⟩ ≡

```
if (testDir[0]) {
  chdir(currentDir);
  if (chdir(testDir) == -1) {
    perror("ctest: test directory"); ABORT;
  }
  getwd(testDir);
  chdir(currentDir);
} else {
  fprintf(stderr, "ctest: Test directory not specified.\n");
  ABORT;
}◇
```

Macro referenced in scrap 15a.

In addition, we want to make sure that we have a full pathname for the `i2c` variable.

⟨Find full pathname of i2c 15d⟩ ≡

```
if (i2c[0] == '\0') {
  strcpy(i2c, "i2c");
} else if (i2c[0] != '/') {
  Char temp[MAXPATHLEN] = "/";

  strcat(temp, i2c);
  strcpy(i2c, currentDir);
  strcat(i2c, temp);
}◇
```

Macro referenced in scrap 15a.

Another case we need to check for occurs when the user specifies the `-R` flag. Since `-R` takes precedence over `-M`, we need to clear the value of `noRun` if `results` is set.

⟨Make `results` and `noRun` consistent 16⟩ ≡

```
if (results == TRUE_STRING) noRun = FALSE_STRING;
```

◇

Macro referenced in scrap 15a.

Chapter 4

Creating the duplicate hierarchy

As stated in the introduction, we root the duplicate hierarchy at a directory named *dir*.*date*, where *dir* is the name of the test directory, and *date* is the current date in the form *YY.MM.DD.hh.mm.ss*. We will store this name in a global called `dupDirLastComponent`. Note that `dupDirLastComponent` is not the full (*absolute*) pathname; it is just the last component of the name.

⟨Global variables 17a⟩ ≡
```
Char dupDirLastComponent[100];
```
◇

Macro defined by scraps 10ac, 17a, 21b.
Macro referenced in scrap 9a.

We create the duplicate hierarchy using the following three steps.

⟨Create the duplicate hierarchy 17b⟩ ≡
  ⟨Create the duplicate directory in a temporary location 17c⟩
  ⟨Start a `gnumake` using `duplicate.script` 19b⟩
  ⟨Prepare the duplicate hierarchy for the test run 21a, … ⟩◇

Macro referenced in scrap 9c.

4.1 Creating the duplicate directory

The first step is to create the duplicate of the test directory, the duplicate directory. We really want to watch out for the case where the user directory is a subdirectory of the test directory, because, if that happens, duplicating the test hierarchy in the duplicate directory will keep copying directories until the filesystem is full.

So, we create the duplicate directory in a temporary location. Our temporary name for the duplicate directory is .*dir*.*date*; it resides in the user directory. Because the name begins with a period, it will be missed by the shell filename globbing facilities (*i.e.,* `*`). Since we locate directories to copy using shell filename globbing, we avoid duplicating the duplicate directory.

Creating the duplicate directory requires two steps for building the name. Then we move to the user directory and create a directory using the name in `dupDirLastComponent`. The second argument to `mkdir` is the permissions mask; it will be ANDed with the user's `umask` variable to generate the permission bits for the directory; we use a word of 1-bits to copy the user's permissions.

⟨Create the duplicate directory in a temporary location 17c⟩ ≡
  ⟨Find the last component of the name of the test directory 18a⟩
  ⟨Append the current date 18c⟩
```
chdir(userDir);
mkdir(dupDirLastComponent, ~0);
```
◇

Macro referenced in scrap 17b.

We find the *.dir* part of the duplicate directory's name by taking the last component of the name of the test directory. We find the last component by working backwards from the end of the test directory's name, stopping when we hit a slash or the beginning of the string. We know that there is not a trailing slash because `getwd` does not append slashes to the names it generates. When we have found the right starting point, we copy the component name, with a dot prepended, into `dupDirLastComponent` using `sprintf`.

⟨Find the last component of the name of the test directory 18a⟩ ≡

```
{
  Char *p = testDir + strlen(testDir);

  while (p != testDir && p[-1] != '/') p--;
  sprintf(dupDirLastComponent, ".%s", p);
}◇
```

Macro referenced in scrap 17c.

Next, we want to put the *.date* part on. To find the current date, we need to call some functions declared in the standard header `<time.h>`.

⟨Include files 18b⟩ ≡

```
#include <time.h>
◇
```

Macro defined by scraps 9b, 10b, 18b.
Macro referenced in scrap 9a.

The `time` function returns the current "calendar" time, whatever that is. We convert calendar time to "local" time using `localtime`. We then generate a string using `strftime`, which uses `printf`-like conversions to generate a string from a local time. In this case, we generate a string `timeString` in the format *.YY.MM.DD.hh.mm.ss*. Then we concatenate the two parts of the name of the duplicate directory using `strcat`.

⟨Append the current date 18c⟩ ≡

```
{
  time_t calTime;
  struct tm *localTime;
  Char timeString[20];

  calTime = time(NULL);
  localTime = localtime(&calTime);
  strftime(timeString, 20, ".%y.%m.%d.%H.%M.%S", localTime);
  strcat(dupDirLastComponent, timeString);
}◇
```

Macro referenced in scrap 17c.

4.2 Starting the gnumake and building the duplicate hierarchy

> **NOTE**. Since the details of programming in `sh` and GNU Makefile are little-known in many circles, we will attempt to provide tutorial documentation when we use obscure features of these systems. Our explanation will accompany the initial use of each feature.

We will use `gnumake` to build the duplicate hierarchy. When we start the `gnumake` process, we will pass it the user's command-line input (as well as other information) using variables set on the `gnumake`'s command line. The `gnumake` process is interested in the following variables:

`TEST_DIR` gives the full pathname of the directory in which to start up the `gnumake` process. We use `testDir`.

`DUP_DIR` is the full pathname of the duplicate directory. We pass in the full pathname of the duplicate directory, constructed from `userDir` and `dupDirLastComponent`.

`SCRIPT` gives the full pathname of the file `duplicate.script`. We reference a `nuweb` constant to provide this value.

To start the `gnumake` process, we need to build a string (using `sprintf`) which we can pass to `sh` using the `system` subroutine. Following is the format string which we'll use in the call to `sprintf`.

⟨Format for building duplicate hierarchy 19a⟩ ≡

```
"gnumake -f %s -C %s TEST_DIR=%s DUP_DIR=%s/%s SCRIPT=%s >/dev/null 2>&1"◇
```

Macro referenced in scrap 19b.

The `-f` switch precedes the name of the file to use as the Makefile. The `-C` switch precedes the name of the directory in which to start the `gnumake` process. The sequence `2>&1` redirects file descriptor 2 (standard error) to be the same as file descriptor 1 (standard output).

Starting the `gnumake` is pretty simple. We supply proper arguments to `sprintf`, making sure that they match the conversions in the format string. Then we pass the resulting `commandString` to `system`, which executes the command using `sh`.

⟨Start a `gnumake` using `duplicate.script` 19b⟩ ≡

```
{
  Char commandString[MAXPATHLEN * 4];

  sprintf(commandString, ⟨Format for building duplicate hierarchy 19a⟩,
          "⟨Full pathname of duplicate.script 33d⟩",
          testDir, testDir, userDir, dupDirLastComponent,
          "⟨Full pathname of duplicate.script 33d⟩");

  printf("Creating duplicate hierarchy....\n");
  /*printf("%s\n", commandString);*/
  system(commandString);
}◇
```

Macro referenced in scrap 17b.

The `gnumake` which we started up in the previous scraps uses `duplicate.script` as its makefile. The first thing this script does is put the names of the test directory's subdirectories in a variable called `SUBDIRS`. Later on, we will decide what to do based on whether there were any subdirectories.

The first line of the following scrap defines `SUBDIRS`. The `:=` symbol evaluates the right-hand side and assigns the result to the variable on the left. By contrast, the `=` symbol would have simply assigned the right-hand side without evaluating it; the contents of the variable would then be evaluated each subsequent time the variable was referenced.

Some text-processing functions are built into `gnumake`. These functions look like variable references; that is, they are of the form `$(`*text*`)`. The `wildcard` function, used here, takes as its argument a pattern, possibly containing wildcard characters. The `wildcard` function then expands the pattern as the C-shell would, attempting to match the pattern to filenames in the current directory. The return value of an invocation of `wildcard` is the list of matching filenames, which is in this case the "dot" file in each of the subdirectories of the current directory.

The output of `wildcard` is passed to the `dir` function, which returns the "directory part" of each filename. In this case it simply strips off the trailing "`.`" characters. This result is assigned to `SUBDIRS`.

"`duplicate.script`" 19c ≡

```
SUBDIRS := $(dir $(wildcard */.))
◇
```

File defined by scraps 19c, 20a.

Once we have found `SUBDIRS`, we can take further action based on the presence or absence of subdirectories. The `ifdef` construct, used here, works in a way similar to that of the C preprocessor's `#ifdef` directive: it masks out part of a file based on the value of a variable. The `else` and `endif` lines complete the construction.

Thus, if there are subdirectories in the current test directory (that is, SUBDIRS has a nonempty value), we will duplicate them; otherwise, we know that the current test directory is a leaf, and we can install the appropriate symbolic links.

"duplicate.script" 20a ≡
```
ifdef SUBDIRS
⟨Duplicate the subdirectories 20b⟩
else
⟨Make symbolic links in the duplicate directory 20d⟩
endif
◇
```
File defined by scraps 19c, 20a.

4.2.1 Nonleaf directories

If the test directory contains subdirectories, we use the following rule for making `all`, the default target. The `for` construction used in this scrap is a feature of `sh`; the minus sign preceding the `for` tells `gnumake` not to abort if the command is unsuccessful.

The `for` statement sets up an iterator, `subdir`, which iterates over all words in `SUBDIRS`. For each `subdir`, we make a copy of the subdirectory in the duplicate directory. Note the `$$` before the use of `subdir`; we do this because we want `subdir` to be referenced by `sh`, not by `gnumake`. The `$$` gets turned into `$` by `gnumake`, which passes `$` on to `sh`.

In addition to copying the subdirectory itself, we also recursively copy the contents of the subdirectory.

⟨Duplicate the subdirectories 20b⟩ ≡
```
all :
        -for subdir in $(SUBDIRS); do \
          mkdir $(DUP_DIR)/$$subdir; \
          ⟨Recursively duplicate the contents of subdir 20c⟩; \
        done◇
```
Macro referenced in scrap 20a.

We recur using another `gnumake` process. The `MAKE` variable is predefined to begin a `gnumake` recursion, retaining all variable values. We use `MAKE` to spawn a recursive process in `subdir`, using `duplicate.script`. We remap `TEST_DIR` to `TEST_DIR/subdir` and `DUP_DIR` to `DUP_DIR/subdir`.

⟨Recursively duplicate the contents of subdir 20c⟩ ≡
```
$(MAKE) \
      -C $(TEST_DIR)/$$subdir \
      -f $(SCRIPT) \
      TEST_DIR=$(TEST_DIR)/$$subdir \
      DUP_DIR=$(DUP_DIR)/$$subdir◇
```
Macro referenced in scrap 20b.

4.2.2 Leaf directories

If, however, the test directory is a leaf, we make symbolic links in the duplicate directory pointing back to each file in the test directory.

⟨Make symbolic links in the duplicate directory 20d⟩ ≡
```
all:
        -ln -sf $(TEST_DIR)/* $(DUP_DIR)◇
```
Macro referenced in scrap 20a.

4.3 Preparing the duplicate hierarchy for the test run

Now that the duplicate hierarchy has been created, we want to give it the name it will carry during the actual test run. We'll do this by renaming the duplicate directory; the new name will be the string pointed to by **dupDirLastComponent + 1**, an expression which just skips over the leading dot.

⟨Prepare the duplicate hierarchy for the test run 21a⟩ ≡

```
{
  Char oldName[MAXPATHLEN];
  Char newName[MAXPATHLEN];

  sprintf(oldName, "%s/%s", userDir, dupDirLastComponent);
  sprintf(newName, "%s/%s", userDir, dupDirLastComponent + 1);
  if (rename(oldName, newName) == -1) perror("ctest:");
}◇
```

Macro defined by scraps 21ac, 22.
Macro referenced in scrap 17b.

Since **dupDirLastComponent** is no longer the name of the duplicate directory, we'll abandon it here in favor of **dupDirFullPath**, which we'll define to be the full pathname of the duplicate directory.

⟨Global variables 21b⟩ ≡

```
Char dupDirFullPath[MAXPATHLEN];
◇
```

Macro defined by scraps 10ac, 17a, 21b.
Macro referenced in scrap 9a.

We use **sprintf** to construct **dupDirFullPath**.

⟨Prepare the duplicate hierarchy for the test run 21c⟩ ≡

```
sprintf(dupDirFullPath, "%s/%s", userDir, dupDirLastComponent + 1);
◇
```

Macro defined by scraps 21ac, 22.
Macro referenced in scrap 17b.

The last detail we need to take care of concerns the log file. We create the test log file in the duplicate directory, prepending a dot to the log file's name. We do this because it is possible that later stages of the testing mechanism will execute an **rm *** in the duplicate directory; we want that command to "miss" the log file.

Then, we log the initial message, giving the parameters of the test. We also print this message to the screen.

⟨Prepare the duplicate hierarchy for the test run 22⟩ ≡

```
{
  File testLogFile;

  chdir(dupDirFullPath);
  testLogFile = fopen(".⟨Test log 33a⟩", "w");
  fprintf(testLogFile, "Test directory:     %s\n", testDir);
  fprintf(testLogFile, "Dupicate directory: %s\n", dupDirFullPath);
  fprintf(testLogFile, "Transformations:    %s\n", transformations);
  fprintf(testLogFile, "I2c:                %s\n", i2c);
  fprintf(testLogFile, "---------------------------------\n");
  fprintf(testLogFile, "Begin ctest run at %s\n", testDir);
  fclose(testLogFile);

  printf("Test directory:     %s\n", testDir);
  printf("Dupicate directory: %s\n", dupDirFullPath);
  printf("Transformations:    %s\n", transformations);
  printf("I2c:                %s\n", i2c);
  printf("---------------------------------\n");
  printf("Begin ctest run at %s\n", testDir);
}◇
```

Macro defined by scraps 21ac, 22.
Macro referenced in scrap 17b.

Chapter 5

Compiling and running test programs

So far we have read in the user's command-line input and created the duplicate hierarchy. This hierarchy is located in the user directory, and its name is stored in `dupDirectoryFullPath`. Each leaf directory in the duplicate hierarchy now contains symbolic links which point to corresponding files in the test hierarchy.

The next task is to start up another `gnumake`, which will actually compile and run the test programs in the duplicate hierarchy. After the `gnumake` finishes, we do some small cleanup chores.

⟨Perform tests 23⟩ ≡
    ⟨Start a `gnumake` using `test.script` 24b⟩
    ⟨Clean up after the test run 30a, ... ⟩
    ◇
Macro referenced in scrap 9c.

5.1 Starting the `gnumake` process

When we start the `gnumake` process, we will pass it the user's command-line input (as well as other information) using variables set on the `gnumake`'s command line. The `gnumake` process is interested in the following variables:

`TRANSFORMATIONS` indicates the transformations to be performed on the ILOC files being compiled for the test. The format for this variable is that which we used to construct `transformations` while parsing the user's command line.

`TEST_DIR` gives the full pathname of the directory in which to start up the `gnumake` process. We use `testDir`.

`DUP_DIR` is the full pathname of the duplicate directory. We pass in `dupDirFullPath`.

`NO_DELETE` may be `"TRUE"` or `"FALSE"`. A `"TRUE"` value means that the test programs should be compiled and run, but not be deleted at the end of processing. We use the value of `noDelete`.

`NO_RUN` may be `"TRUE"` or `"FALSE"`. A `"TRUE"` value means that the test programs should be compiled, but they should not be run, nor should they be deleted at the end of processing. We use the value of `noRun`.

`RESULTS` is another "Boolean" value which may be either `"TRUE"` or `"FALSE"`. A `"TRUE"` value tells `gnumake` that we are building the `Results` files instead of running tests.

`QUIET` is yet another "Boolean" value; a `"TRUE"` value tells the test mechanism not to print status messages to `/dev/tty`.

`I2C` gives the pathname for the `i2c` translator for this test run. We pass in the contents of the global variable `i2c`.

`F2I_FLAGS` gives the flags to be passed to `f2i`.

I2C_FLAGS gives the flags to be passed to i2c.

SCRIPT gives the full pathname of the file test.script. We pass in a value defined by a nuweb constant.

TEST_LOG is the name of the test log file. We use the name of the test log with a dot prepended.

We incorporate the above elemements into a Bourne shell (sh) command string using the following sprintf format. Note the redirection of the standard output and standard error; this redirection is what provides the make log—the record of the commands issued by the gnumake.

⟨Format for compiling and running tests 24a⟩ ≡

```
"gnumake -f %s -C %s \
       \"TRANSFORMATIONS=%s\" TEST_DIR=%s DUP_DIR=%s \
       NO_DELETE=%s NO_RUN=%s RESULTS=%s QUIET=%s TEST_LOG=%s/.%s \
       I2C=%s F2I_FLAGS=%s I2C_FLAGS=%s SCRIPT=%s \
       >> %s/.%s 2>&1"◇
```

Macro referenced in scrap 24b.

We use the above format string here in a sprintf call; this call creates the actual command string by substituting paramaters in for the %s tokens in the format string. When the command string has been built, we use system to pass the command string to the Bourne shell (sh).

⟨Start a gnumake using test.script 24b⟩ ≡

```
{
  Char commandString[MAXPATHLEN * 4];

  sprintf(commandString, ⟨Format for compiling and running tests 24a⟩,
          "⟨Full pathname of test.script 33e⟩", dupDirFullPath,
          transformations, testDir, dupDirFullPath,
          noDelete, noRun, results, quietMode, dupDirFullPath, "⟨Test log 33a⟩",
          i2c, f2iFlags, i2cFlags, "⟨Full pathname of test.script 33e⟩",
          dupDirFullPath, "⟨Make log 33b⟩");

/*   printf("%s\n", commandString); */
  system(commandString);
}◇
```

Macro referenced in scrap 23.

5.2 gnumake processing

The gnumake process starts up in the duplicate directory, using the makefile test.script. We begin test.script by defining a variable for handling log file output. We define LOG to be a command which prints its input to the screen and to the test log file, unless QUIET is defined, in which case the LOG command simply sends its input to the test log file.

"test.script" 24c ≡

```
ifndef QUIET
LOG = tee /dev/tty >> $(TEST_LOG)
else
LOG = cat >> $(TEST_LOG)
endif
◇
```

File defined by scraps 24c, 25c.

In addition, the following macros define output which occurs on entrance or exit from a directory; invocations of these macros will bracket whatever work we do in a directory. The @ sign preceding the command tells gnumake not to echo the command into MAKE.LOG.

⟨Enter-directory message 25a⟩ ≡
```
@echo "In directory      " $(TEST_DIR) | $(LOG)◇
```
Macro referenced in scraps 25d, 26c.

⟨Exit-directory message 25b⟩ ≡
```
@echo "Finished directory" $(TEST_DIR) | $(LOG)◇
```
Macro referenced in scraps 25d, 26c.

Before we can begin processing a directory, we have to initialize a variable called `SUBDIRS` to contain the names of all of the subdirectories of the current directory. Since leaf directories do not contain subdirectories, we can decide whether we are in a leaf directory or in a non-leaf directory by testing the value of `SUBDIRS`.

"test.script" 25c ≡
```
SUBDIRS := $(dir $(wildcard */.))
ifdef SUBDIRS
⟨Non-leaf directory processing 25d⟩
else
⟨Leaf directory processing 26a, ... ⟩
endif◇
```
File defined by scraps 24c, 25c.

5.2.1 Processing for non-leaf directories

The first action to take in a non-leaf directory is to print a status message to the log file and screen. Then, for each directory `subdir` in `SUBDIRS`, we spawn off a recursive `gnumake` in `subdir`. When the `gnumake` is finished, we attempt to remove `subdir`. If all of the test programs in `subdir` were successful, then `subdir` will be empty and the `rmdir` command will be able to remove the directory. Otherwise, `rmdir` cannot remove the directory. The "`-`" before the `rmdir` command tells `gnumake` not to abort if the `rmdir` command fails. When we have finished all subdirectory processing, we print exit messages to the log file and screen.

⟨Non-leaf directory processing 25d⟩ ≡
```
all :
        ⟨Enter-directory message 25a⟩
        -for subdir in $(SUBDIRS); do \
          ⟨Spawn a recursive gnumake in subdir 25e⟩; \
          rmdir $$subdir; \
        done
        ⟨Exit-directory message 25b⟩◇
```
Macro referenced in scrap 25c.

We start up the recursive `gnumake`s using the `MAKE` variable. This variable is automatically defined by `gnumake` for use in recursive situations like this one. A `gnumake` started up by a use of `MAKE` preserves all variables defined in the parent `gnumake` process. However, we want to override some of these variables; we can do this by setting those variables on the command line.

We want our recursive `gnumake` to run in `subdir`, so we use the `-C` option. We want to read from the same makefile, `test.script`, so we use `-f`. As was stated in Section 1.2.2, we remap `TEST_DIR` to the `subdir` in the test hierarchy and `DUP_DIR` to the `subdir` in the duplicate hierarchy.

⟨Spawn a recursive gnumake in subdir 25e⟩ ≡
```
$(MAKE) \
      -C $$subdir \
      -f $(SCRIPT) \
      TEST_DIR=$(TEST_DIR)/$$subdir \
      DUP_DIR=$(DUP_DIR)/$$subdir◇
```
Macro referenced in scrap 25d.

5.2.2 Processing for leaf directories

If we are in a leaf directory, we need to include the `Ctest.script` file. If that file does not exist in the directory, the `gnumake` will terminate. If this is not the top-level `gnumake`, processing will then continue on other directories.

⟨Leaf directory processing 26a⟩ ≡

```
include $(TEST_DIR)/Ctest.script
◇
```

Macro defined by scraps 26abc, 28ab.
Macro referenced in scrap 25c.

Before we begin further processing of the leaf directory, we define a variable, `ERR`, which will echo the name of the directory, followed by an error message, supplied by the user at the site of the reference to `ERR`. We will use this variable to report errors, performing something similar to the following:

```
$(ERR) Fortran compiler error
```

To find the name of the current directory, we use the `shell` function to execute the `pwd` command, and we use `notdir` to remove all but the last component of the string returned by `pwd`.

⟨Leaf directory processing 26b⟩ ≡

```
ERR := echo ERROR: $(notdir $(shell pwd)):
◇
```

Macro defined by scraps 26abc, 28ab.
Macro referenced in scrap 25c.

The `all` target

The default target for this `gnumake` is `all`. This target depends on `main`, which is the executable test program itself. We will present a rule for `main` shortly.

Once `main` has been made, we can make `all`. The commands for making `all` are the ones which actually run the test program and compare its output to `Results`.

If the variable `NO_RUN` has been set, we do nothing. Since `NO_RUN` corresponds to the user's `-M` command-line option, all we needed to do was make `main`. That finished, we exit.

Otherwise, we run the program, saving its output in `Newresults`. Then if `RESULTS` is set (by the `-R` option), we copy `Newresults` into the test hierarchy as `Results`. Otherwise we compare `Newresults` against `Results`.

Then, if `NO_DELETE` is not set, we attempt to remove all the files in `DUP_DIR`. Finally, we log exit messages.

⟨Leaf directory processing 26c⟩ ≡

```
all :
        ⟨Enter-directory message 25a⟩
        ⟨Make main 27a⟩
ifndef NO_RUN
        ⟨Run the program 27b⟩
ifdef RESULTS
        ⟨Copy the Newresults file into the test hierarchy 27c⟩
else
        ⟨Compare Newresults to Results 27d⟩
endif
ifndef NO_DELETE
        ⟨Remove all files if there was no error 27e⟩
endif
endif
        ⟨Exit-directory message 25b⟩
◇
```

Macro defined by scraps 26abc, 28ab.
Macro referenced in scrap 25c.

⟨Make main 27a⟩ ≡

```
$(MAKE) -f $(SCRIPT) main
◇
```

Macro referenced in scrap 26c.

We run the program by issuing the command `./main`, including on the command line the `INPUT` string specified in `Ctest.script`. We prepend a `-` to the command so that the `gnumake` will not terminate if the program does not return a zero exit value. We save the standard output and standard error in `Newresults`, then append to `Newresults` any file specified in the `OUTPUT` variable, (which the user sets in `Ctest.script`). Note that we include `/dev/null` in the `cat` command; we do this so that `cat` will not try to read the standard input if there is no `OUTPUT`.

⟨Run the program 27b⟩ ≡

```
echo "  Running main" | $(LOG)
-./main $(INPUT) > Newresults # 2>&1
cat $(OUTPUT) /dev/null >> Newresults◇
```

Macro referenced in scrap 26c.

We copy the `Newresults` file into the test hierarchy in the obvous manner.

⟨Copy the `Newresults` file into the test hierarchy 27c⟩ ≡

```
-cp -f Newresults $(TEST_DIR)/Results◇
```

Macro referenced in scrap 26c.

We compare `Newresults` and `Results` in three steps. First, we check for the presence of the `Results` file; if it does not exist we write error messages to the log and to the `ERROR` file. We will use the `ERROR` file later on; its presence indicates that an error occured somewhere in this directory.

If `Results` exists, we use `diff` to do the comparison, storing the `diff` output in the file `DIFF`. Then, if `DIFF` has nonzero length, we issue error messages to the log and `ERROR`.

⟨Compare `Newresults` to `Results` 27d⟩ ≡

```
if test ! -f Results; then \
  $(ERR) Results file not found | tee -a ERROR | $(LOG); \
fi
-diff Results Newresults > DIFF
if test -s DIFF; then \
  $(ERR) Results did not match | tee -a ERROR | $(LOG); \
fi◇
```

Macro referenced in scrap 26c.

Finally, if there was no error in this directory, we remove all files from it. We can test for errrors by checking for the existence of the `ERROR` file.

The `test` command returns true or false values depending on its arguments. The `-f ERROR` sequence tells `test` to return zero (true) if the file `ERROR` exists. However, the `!` is `test`'s negation operator, so this `test` command returns true is `ERROR` does not exist.

⟨Remove all files if there was no error 27e⟩ ≡

```
if test ! -f ERROR ; then rm *; fi◇
```

Macro referenced in scrap 26c.

The `main` target

We make the executable program `main` from object files; we make the object files are made from ILOC and Fortran files. Here we define a variable called `OBJS` to contain a list of filenames ending in `.o`; each `.o` file in `OBJS` corresponds to a `.i` or `.f` file in the directory.

We construct `OBJS` using the `patsubst` and `wildcard` functions. We use `wildcard` with either `*.i` or `*.f` as the pattern. Our invocations of `patsubst` replace (for instance) all occurances of `%.i` with `%.o`, where `%`

is a wildcard which matches any character sequence. The `patsubst` function does its processing on its third argument.

⟨Leaf directory processing 28a⟩ ≡
```
OBJS := $(patsubst %.i,%.o,$(wildcard *.i)) $(patsubst %.f,%.o,$(wildcard *.f))
```
◇

Macro defined by scraps 26abc, 28ab.
Macro referenced in scrap 25c.

There are two ways to make `main`. The first method treats all Fortran files as drivers, compiling them with `xlf`. ILOC files are run through the user-specified tranformations and compiled with `i2c` and `xlc`.

The second method uses `f2i` to convert Fortran to ILOC, then uses transformations to optimize the ILOC, then uses `i2c` and `xlc` to produce object code.

We use the first method if we are building `Results` or if there are any ILOC files present. We use the second method in other cases.

⟨Leaf directory processing 28b⟩ ≡
```
ifdef RESULTS
⟨Build main using xlf 28c, ... ⟩
else
ILOC_SRC := $(wildcard *.i)
ifdef ILOC_SRC
⟨Build main using xlf 28c, ... ⟩
else
⟨Build main using f2i 29b, ... ⟩
endif
endif◇
```

Macro defined by scraps 26abc, 28ab.
Macro referenced in scrap 25c.

First method

The following three code scraps comprise the first method of making `main`. This scrap shows the linking step, where we use `xlf` to link together all files ending in `.o`. Notice that, although `main` *depends* on `OBJS`, we do not *construct* `main` from `OBJS`; we use `*.o` instead. This strategy allows us to precompile some files, saving compilation time.

⟨Build main using xlf 28c⟩ ≡
```
main : $(OBJS)
        echo "  Linking main" | $(LOG)
        -xlf *.o -o main
```
◇

Macro defined by scraps 28cd, 29a.
Macro referenced in scrap 28b.

We use the following rule to build object files from ILOC files. The rule's commands run an ILOC file through the user-specified transformations, then through `i2c`. Finally, the resulting C file is compiled with `xlc`. Errors are reported using the `ERR` and `LOG` variables, as usual.

⟨Build main using xlf 28d⟩ ≡
```
%.o : %.i
        echo "  Transforming" $*.i | $(LOG)
        -cat $< $(TRANSFORMATIONS) | $(I2C) $(I2C_FLAGS) > $*.c \
            || $(ERR) Transformation or i2c error | tee -a ERROR | $(LOG)
        echo "  Compiling" $*.c | $(LOG)
        -xlc -O -c -qspill=3000 $*.c || $(ERR) C compiler error | tee -a ERROR | $(LOG)
```
◇

Macro defined by scraps 28cd, 29a.
Macro referenced in scrap 28b.

We make an object file from a Fortran file using the following rule, which uses **xlf** to compile a **.f** file. We report errors in the usual manner.

It is significant that this rule follows the rule which compiles ILOC files, because **gnumake** applies the rules in the order in which they are found. Our arrangement of the rules allows ILOC files to take precedence over their corresponding Fortran files.

⟨Build `main` using `xlf` 29a⟩ ≡

```
%.o : %.f
        echo "  Compiling" $*.f | $(LOG)
        -xlf -O -c $< || echo Fortran compiler error | tee -a ERROR | $(LOG)
```

◇

Macro defined by scraps 28cd, 29a.
Macro referenced in scrap 28b.

Second method

The following two scraps comprise the second method of making **main**. This scrap is the link step, which links ***.o** together to form **main**. This step differs from it first-method counterpart in that it also compiles and links in Fortran files with names ending in **.io.f**. These files, possibly not present in the test directory, can be created by **f2i** (see the next scrap).

⟨Build `main` using `f2i` 29b⟩ ≡

```
main : $(OBJS)
        echo "  Linking main" | $(LOG)
        -xlf *.o *.io.f -o main
```

◇

Macro defined by scraps 29bc.
Macro referenced in scrap 28b.

We create ILOC files from the Fortran files using **f2i**, then pass the ILOC through **i2c** to generate C. We log errors as usual, but we also have to erase a directory named **PARA**, which **f2i** creates during its processing. In addition, **f2i** may create files whose names end in **.io.f**; we will leave these files to be compiled by the link step.

We compile the C files using **xlc**, generating object files which will be linked by **xlf** in the link step.

⟨Build `main` using `f2i` 29c⟩ ≡

```
%.o : %.f
        echo "  Transforming" $*.f | $(LOG)
        -f2i $(F2I_FLAGS) -M $< $(TRANSFORMATIONS) \
          | $(I2C) $(I2C_FLAGS) > $*.c \
          || $(ERR) f2i, transformation or i2c error | tee -a ERROR | $(LOG)
        -rm -fr PARA
        echo "  Compiling" $*.c | $(LOG)
        -xlc -O -c -qspill=3000 $*.c || $(ERR) C compiler error | tee -a ERROR | $(LOG)
```

◇

Macro defined by scraps 29bc.
Macro referenced in scrap 28b.

5.3 Concluding the test run

The cleanup details we need to take care of concern the log file. We need to log the closing message to the test log and to the screen.

⟨Clean up after the test run 30a⟩ ≡

```
{
  File testLogFile;

  chdir(dupDirFullPath);
  testLogFile = fopen(".⟨Test log 33a⟩", "a");
  fprintf(testLogFile, "End ctest run\n");
  printf("End ctest run\n");
  fclose(testLogFile);
}◇
```

Macro defined by scraps 30abc.
Macro referenced in scrap 23.

Next, we `grep` the test log for error reports, spitting the resulting summary out to the screen.

⟨Clean up after the test run 30b⟩ ≡

```

  printf("---------------------------------\n");
  printf("The following errors were found:\n");
  system("grep ERROR .⟨Test log 33a⟩");
◇
```

Macro defined by scraps 30abc.
Macro referenced in scrap 23.

When all testing is finished, we need to rename the log files, removing the prepended dots from their file names. This is similar to the renaming of the duplicate directory which occurs after the construction of the duplicate hierarchy.

⟨Clean up after the test run 30c⟩ ≡

```
{
  Char oldName[MAXPATHLEN];
  Char newName[MAXPATHLEN];

  sprintf(oldName, "%s/.⟨Test log 33a⟩", dupDirFullPath);
  sprintf(newName, "%s/⟨Test log 33a⟩", dupDirFullPath);
  rename(oldName, newName);

  sprintf(oldName, "%s/.⟨Make log 33b⟩", dupDirFullPath);
  sprintf(newName, "%s/⟨Make log 33b⟩", dupDirFullPath);
  rename(oldName, newName);
}◇
```

Macro defined by scraps 30abc.
Macro referenced in scrap 23.

Appendix A

Indices

A.1 Files

A.2 Macros

A.3 Identifiers

A.4 Constants

This constant gives the name of the test log file.

⟨Test log 33a⟩ ≡
    `TEST.LOG`◇
Macro referenced in scraps 22, 24b, 30abc.

This is the name of the make log file.

⟨Make log 33b⟩ ≡
    `MAKE.LOG`◇
Macro referenced in scraps 24b, 30c.

This is the directory in which we expect to find `duplicate.script` and `test.script`.

⟨Ctest home directory 33c⟩ ≡
    `/compiler/tools/ctest`◇
Macro referenced in scraps 33de.

This is the full pathname of `duplicate.script`.

⟨Full pathname of `duplicate.script` 33d⟩ ≡
    ⟨Ctest home directory 33c⟩`/duplicate.script`◇
Macro referenced in scrap 19b.

This is the full pathname of `test.script`.

⟨Full pathname of `test.script` 33e⟩ ≡
    ⟨Ctest home directory 33c⟩`/test.script`◇
Macro referenced in scrap 24b.

A.5 Function prototypes

`AddTransformation` takes two arguments: `name`, a string naming a transformation, and `currChar`, a pointer to the current location in the current `argVector` entry. `AddTransformation` inserts the transformation name into the `transformations` string, followed by whatever characters it finds enclosed in square brackets in the `argVector` entry. `AddTransformation` returns the position in the `argVector` entry up to which it read.

⟨Prototypes 33f⟩ ≡
```
Char *AddTransformation(Char *name, Char *currChar);
```
    ◇
Macro referenced in scrap 9a.