CSE306 Software Quality in Practice

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Review

GNU make

text and examples from gmake documentation

https://www.gnu.org/software/make/manual/make.html

- So far:
  - rules, targets, prerequisites, recipes
  - explicit and implicit rules
  - makefile naming
  - phony targets
  - variables
  - comments
  - regular and order-only prerequisites

4.4 Wildcards

https://www.gnu.org/software/make/manual/make.html#Wildcards

- The wildcard characters in make are '*', '?' and '[…]'
- Wildcard expansion is performed by make automatically in targets and in prerequisites.
- In recipes, the shell is responsible for wildcard expansion.
- In other contexts, wildcard expansion happens only if you request it explicitly with the wildcard function.

Suppose you would like to say that the executable file foo is made from all the object files in the directory, and you write this:

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

The value of objects is the actual string '*.o'. Wildcard expansion happens in the rule for foo, so that each existing '.o' file becomes a prerequisite of foo and will be recompiled if necessary.

But what if you delete all the '.o' files? When a wildcard matches no files, it is left as it is, so then foo will depend on the oddly-named file *.o. Since no such file is likely to exist, make will give you an error saying it cannot figure out how to make *.o.

4.4.3 The Function wildcard

Wildcard expansion happens automatically in rules. But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function. If you want to do wildcard expansion in such places, you need to use the `wildcard` function, like this:

```
$(wildcard pattern…)
```

This string, used anywhere in a makefile, is replaced by a space-separated list of names of existing files that match one of the given file name patterns. If no existing file name matches a pattern, then that pattern is omitted from the output of the `wildcard` function. Note that this is different from how unmatched wildcards behave in rules, where they are used verbatim rather than ignored (see Wildcard Pitfall).

One use of the `wildcard` function is to get a list of all the C source files in a directory, like this:

```
$(wildcard *.c)
```

We can change the list of C source files into a list of object files by replacing the '`.c`' suffix with '`.o`' in the result, like this:

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

(Here we have used another function, `patsubst`. See Functions for String Substitution and Analysis.)

Thus, a makefile to compile all C source files in the directory and then link them together could be written as follows:

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(objects)
```

(This takes advantage of the implicit rule for compiling C programs, so there is no need to write explicit rules for compiling the files. See The Two Flavors of Variables, for an explanation of '`:=`', which is a variant of '`=`'.)

4.5.1 VPATH

https://www.gnu.org/software/make/manual/make.html#Directory-Search

- The value of the make variable VPATH specifies a list of directories that make should search. Most often, the directories are expected to contain prerequisite files that are not in the current directory; however, make uses VPATH as a search list for both prerequisites and targets of rules.
- Example: VPATH = src:../headers

4.5.2 VPATH

https://www.gnu.org/software/make/manual/make.html#Directory-Search

- Similar to the VPATH variable, but more selective, is the vpath directive (note lower case), which allows you to specify a search path for a particular class of file names: those that match a particular pattern. Thus you can supply certain search directories for one class of file names and other directories (or none) for other file names.
- A vpath pattern is a string containing a '%' character. The string must match the file name of a prerequisite that is being searched for, the '%' character matching any sequence of zero or more characters (as in pattern rules; see Defining and Redefining Pattern Rules). For example, %.h matches files that end in .h.
- Example: vpath %.h ../headers

4.14 Generating Prerequisites Automatically

https://www.gnu.org/software/make/manual/make.html#Automatic-Prerequisites

To avoid this hassle, most modern C compilers can write these rules for you, by looking at the #include lines in the source files. Usually this is done with the ‘-M’ option to the compiler. For example, the command:

```
cc -M main.c
```

Use -MM instead of -M to "[omit] prerequisites on system header files."

generates the output:

```
main.o : main.c defs.h
```

Thus you no longer have to write all those rules yourself. The compiler will do it for you.

Section 4.14 goes into some detail on how to set up rules to automatically create the dependencies. That is more involved than I expect you to learn in this course. Instead, run

```
gcc -MM file.c
```

manually to get the prerequisites right; insert them directly into the makefile.

5.1 Recipe Syntax

https://www.gnu.org/software/make/manual/make.html#Recipe-Syntax

- Two kinds of syntax used in a makefile:
  - Recipes use shell syntax
  - Rest of makefile uses make syntax

- This has important implications for how variables are interpreted

Example

https://www.gnu.org/software/make/manual/make.html#Variables-in-Recipes

Rule in makefile

```
LIST = one two three
all:
        for i in $(LIST); do \
            echo $$i; \
        done
```

Example
https://www.gnu.org/software/make/manual/make.html#Variables-in-Recipes

What the shell sees

```
for i in one two three; do \
    echo $i; \
done
```

Example

https://www.gnu.org/software/make/manual/make.html#Variables-in-Recipes

The output produced

```
one
two
three
```

Automatic variables

https://www.gnu.org/software/make/manual/make.html#Automatic-Variables

$@ The file name of the target of the rule.

$% The target member name, when the target is an archive member.

$< The name of the first prerequisite.

$? The names of all the prerequisites that are newer than the target, with spaces between them.

$^ The names of all the prerequisites, with spaces between them.

$+ This is like ‘$^’, but prerequisites listed more than once are duplicated in the order they were listed in the makefile.

$| The names of all the order-only prerequisites, with spaces between them.

$* The stem with which an implicit rule matches (see How Patterns Match).

Lecture question

Automatic variables

https://www.gnu.org/software/make/manual/make.html#Automatic-Variables

‘$(@D)’
The directory part of the file name of the target, with the trailing slash removed. If the value of ‘$@’ is dir/foo.o then ‘$(@D)’ is dir. This value is . if ‘$@’ does not contain a slash.
‘$(@F)’
The file-within-directory part of the file name of the target. If the value of ‘$@’ is dir/foo.o then ‘$(@F)’ is foo.o. ‘$(@F)’ is equivalent to ‘$(notdir $@)’.

‘$(*D)’
‘$(*F)’
The directory part and the file-within-directory part of the stem; dir and foo in this example.

‘$(%D)’
‘$(%F)’
The directory part and the file-within-directory part of the target archive member name. This makes sense only for archive member targets of the form archive(member) and is useful only when member may contain a directory name. (See Archive Members as Targets.)

‘$(<D)’
‘$(<F)’
The directory part and the file-within-directory part of the first prerequisite.

‘$(^D)’
‘$(^F)’
Lists of the directory parts and the file-within-directory parts of all prerequisites.

‘$(+D)’
‘$(+F)’
Lists of the directory parts and the file-within-directory parts of all prerequisites, including multiple instances of duplicated prerequisites.

‘$(?D)’
‘$(?F)’
Lists of the directory parts and the file-within-directory parts of all prerequisites that are newer than the target.

5.3 Recipe execution

When it is time to execute recipes to update a target, they are executed by invoking a new sub-shell for each line of the recipe, unless the .ONESHELL special target is in effect

5.3 Recipe execution

Sometimes you would prefer that all the lines in the recipe be passed to a single invocation of the shell.

```
.ONESHELL:
SHELL = /usr/bin/perl
.SHELLFLAGS = -e
show :
        # Make sure "@" is not the first character on the first line
        @f = qw(a b c);
        print "@f\n";
```

What OS are you running on?

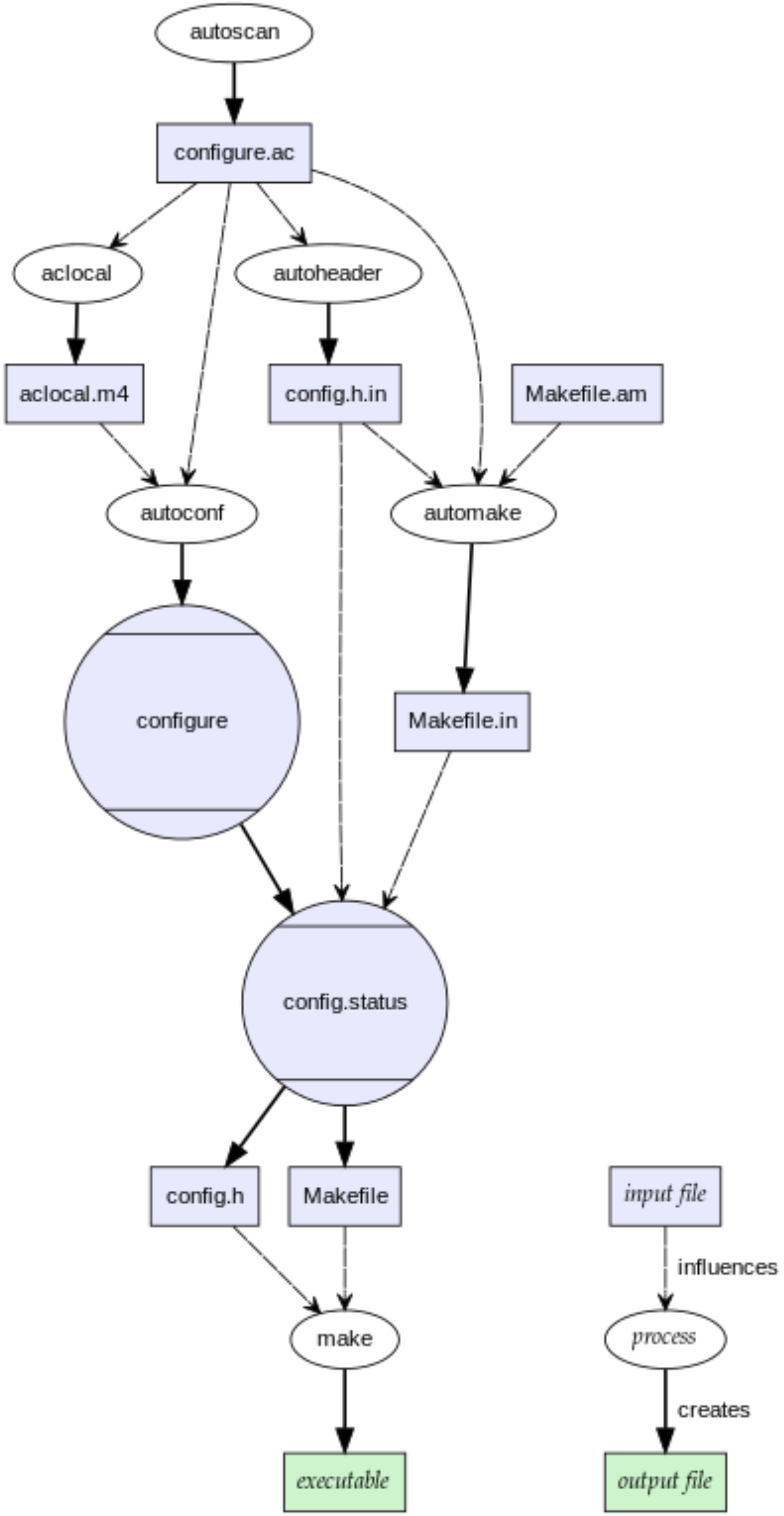
OS identification from
https://stackoverflow.com/questions/714100/os-detecting-makefile

- uname -s

```
OS = $(shell uname -s)

ifeq ($(OS),Darwin)
  MSG = You are running macos
else ifeq ($(OS),Linux)
  MSG = You are running linux
else
  MSG = You are running something else
endif
```

CC BY-SA 3.0
Author: Jdthood
Source: en.wikipedia.org/wiki/GNU_Build_System#/media/File:Autoconf-automake-process.svg



Review this lecture and last
before LEX 11

opaque vs transparent testing

opaque testing

- Can anyone describe what this is?

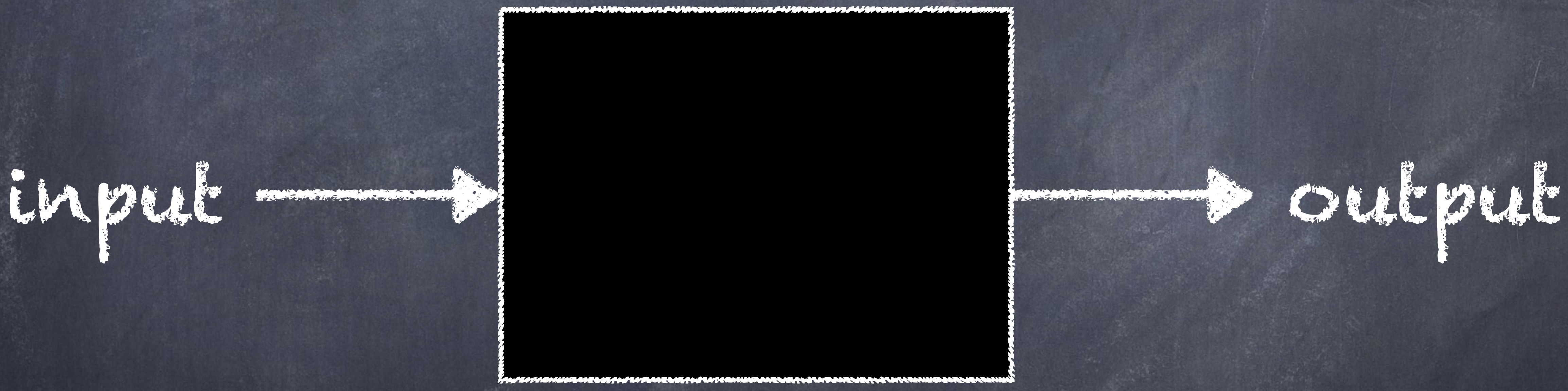
opaque testing

Testing done to probe the input/output behavior of a system, without knowledge of the interior structure.

TDD relies on opaque testing to capture requirements of a software component before it is implemented.

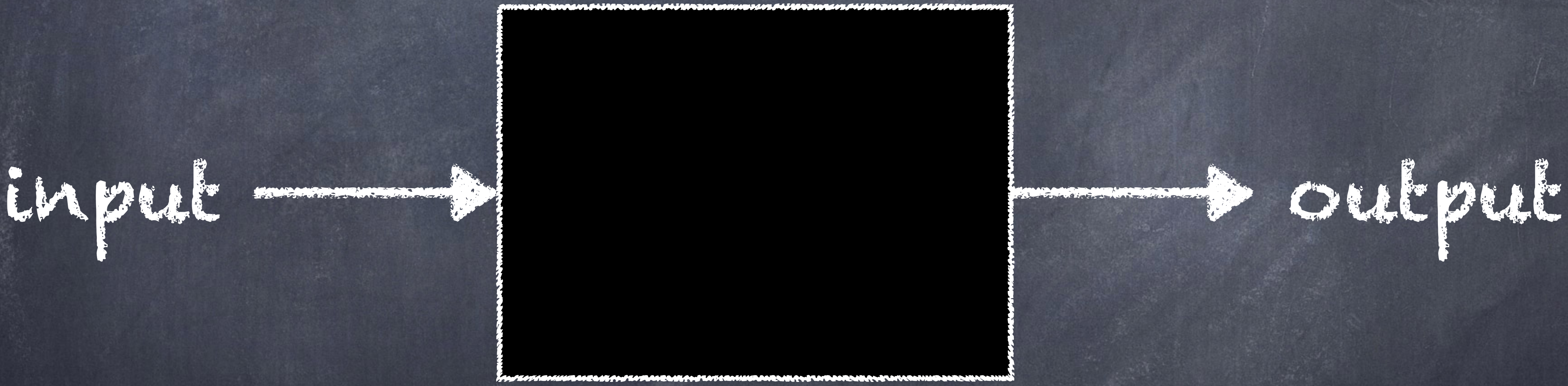
opaque testing

- Code is treated as a closed box, one which you cannot peek inside



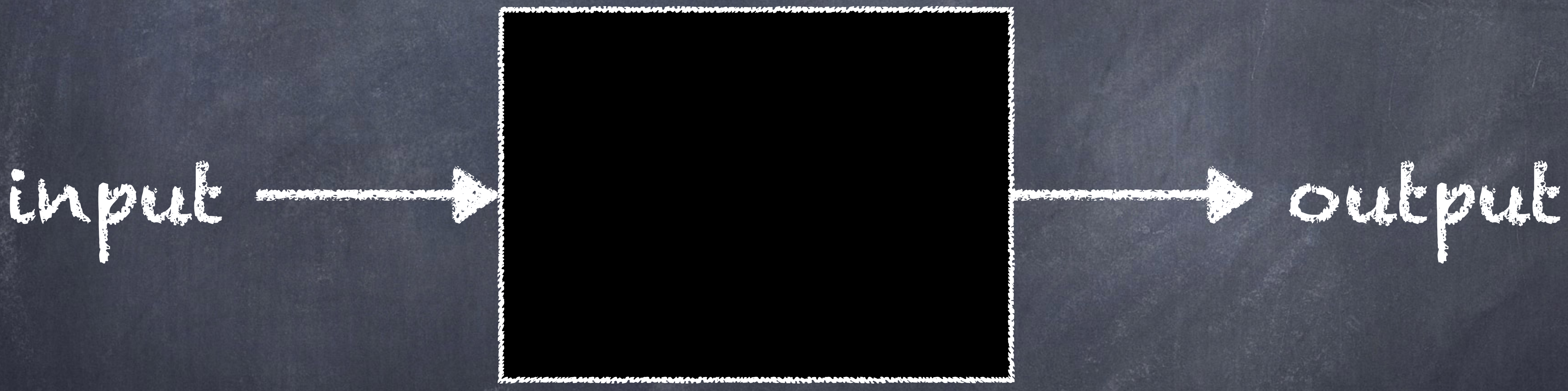
opaque testing

- Tests are written without regard to HOW code is written



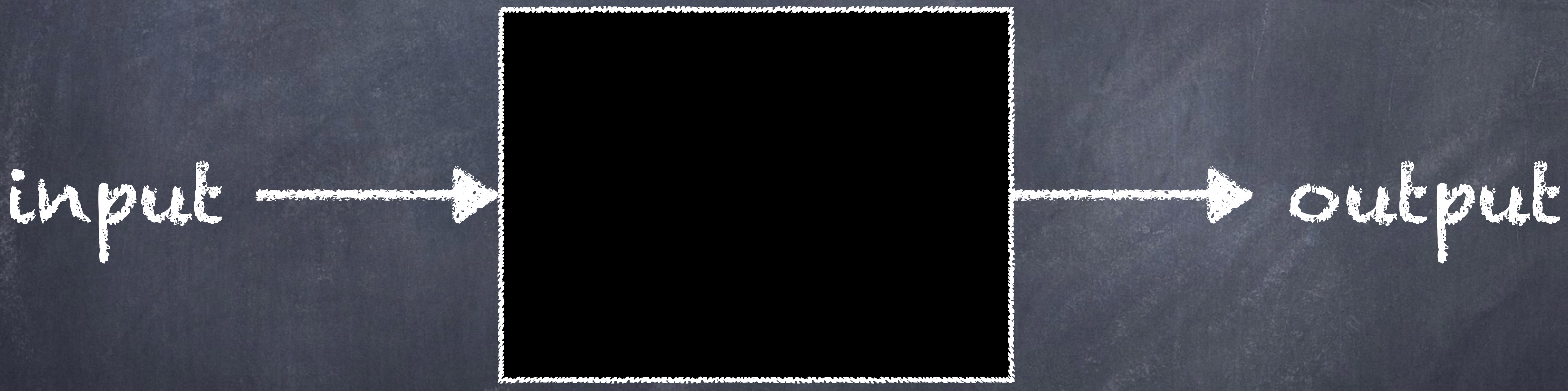
opaque testing

- Tests are meant to capture the intended behavior of the system (the requirements/ specifications): WHAT the code should do.



opaque testing

- In Test Driven Development (TDD) tests are written before the code is, and so qualifies as opaque testing.

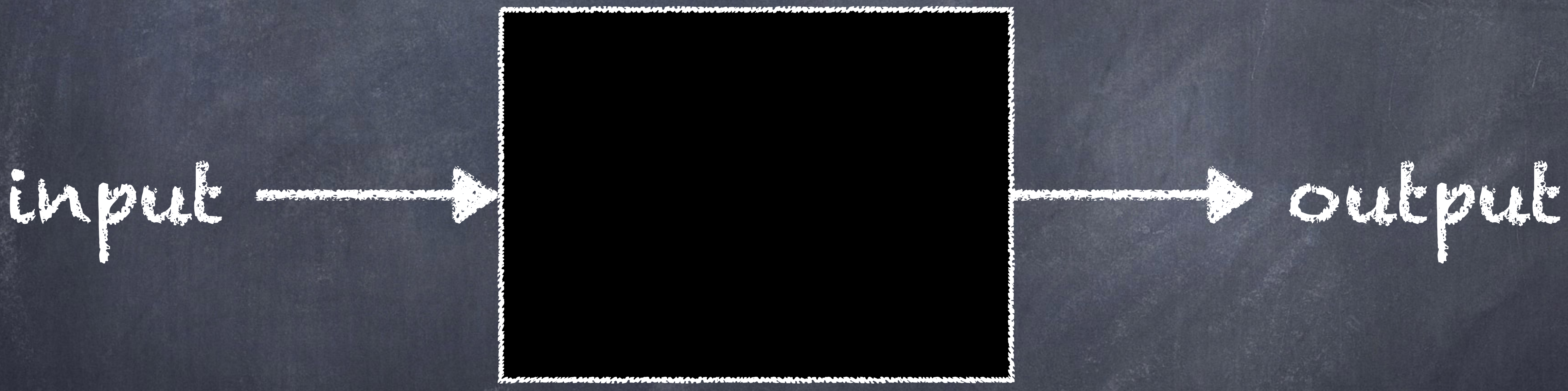


input

output

opaque testing

- In TDD, think of tests written to capture specifications as executable specifications.



transparent testing

- Can anyone describe what this is?

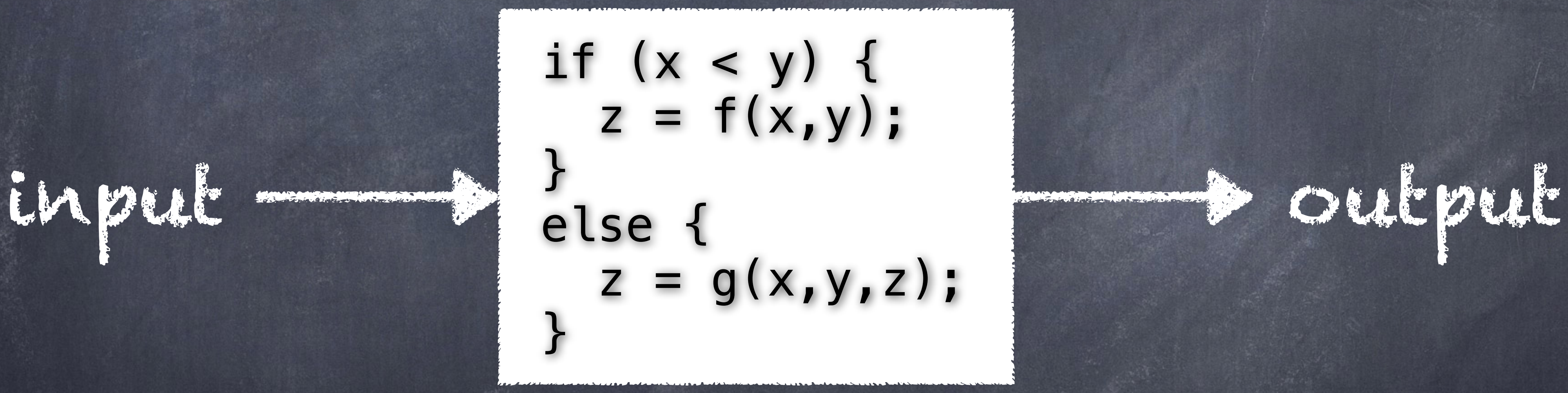
transparent testing

Testing done to probe the input/output behavior of a system, knowing knowledge of the interior structure.

TDD relies on transparent testing to ensure that all computation paths of a software component implementation are covered by test cases.

transparent testing

- Tests are written taking into consideration **HOW** the code is written.



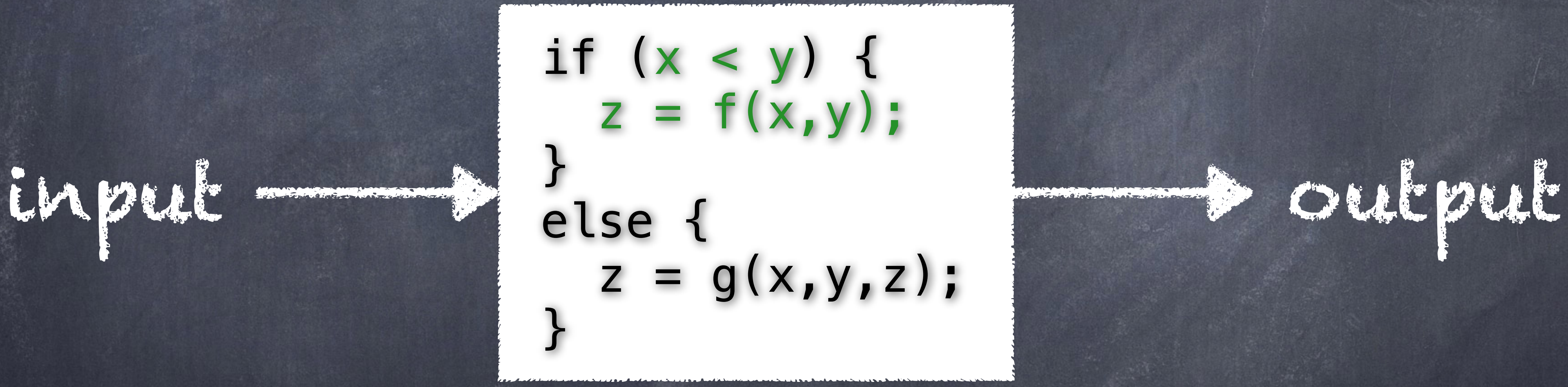
input

```
if (x < y) {
  z = f(x,y);
}
else {
  z = g(x,y,z);
}
```

output

transparent testing

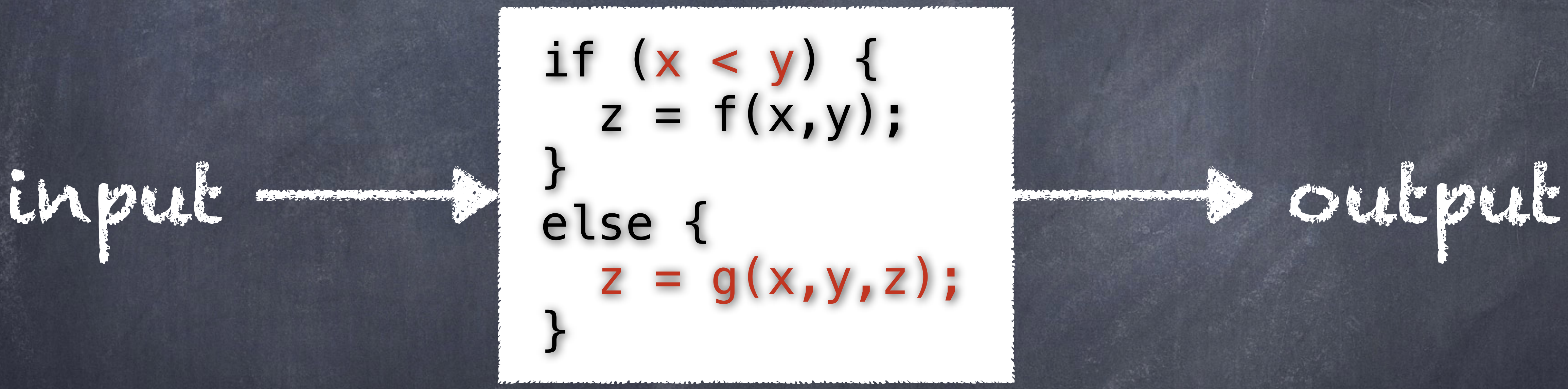
- Use a code coverage tool to ensure that tests exercise ALL possible computation paths.



```
if (x < y) {
  z = f(x,y);
}
else {
  z = g(x,y,z);
}
```

transparent testing

- Use a code coverage tool to ensure that tests exercise ALL possible computation paths.



```
if (x < y) {
  z = f(x,y);
}
else {
  z = g(x,y,z);
}
```

Code coverage

- We will use gcov as our coverage tool.
- Compile with,

```
-fprofile-arcs
-ftest-coverage
-lgcov
```

- as in:

```
gcc $(CFLAGS) -fprofile-arcs -ftest-coverage
              -L /util/CUnit/lib
              -I /util/CUnit/include/CUnit/
              $(OBJECTS) tests.c -o tests
              -lcriterion -lgcov
```

`-fprofile-arcs`

Instrument arcs during compilation. For each function of your program, GCC creates a program flow graph, then finds a spanning tree for the graph.

https://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_2.html#SEC9

`-ftest-coverage`

Create data files for the gcov code-coverage utility (see section gcov: a GCC Test Coverage Program).

https://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_2.html#SEC9

`-llibrary`

Search the library named `library` when linking.
It makes a difference where in the command you write this option; the linker searches/processes libraries and object files in the order they are specified. Thus,

```
foo.o -lz bar.o
```

searches library `z' after file `foo.o' but before `bar.o'. If `bar.o' refers to functions in `z', those functions may not be loaded.
[...]
The directories searched include several standard system directories plus any that you specify with `-L'.

https://gcc.gnu.org/onlinedocs/gcc-2.95.2/gcc_2.html#SEC13

using gcov to verify test coverage

- compile test code with extra flags
    - this instruments code to gather coverage information
- run tests
    - this runs your tests and allows the instrumentation to collect coverage data that shows what parts of the implementation were exercised by the tests
- run gcov on the source file (e.g. source.c) whose coverage you're interested in exploring
- use 'man gcov' to see gcov command line options. Try -b.
- Look at the file produced by gcov (e.g. source.c.gcov)