

Part I: Test Coverage

Test Coverage

How do you know whether a program is tested well?

- Data coverage (HtDP)
 - try every variant
 - try to get all results
- Code coverage

Test Coverage

How do you know whether a program is tested well?

- Data coverage (HtDP)
 - try every variant
 - try to get all results
- ➔ • Code coverage

Test Coverage

function — every function called

✓

```
int max(int n, int m) {  
    if (n > m)  
        return n;  
    else  
        return m;  
}
```

✓

```
int maxabs(int n, int m) {  
    int absn = ((n < 0) ? -n : n);  
    int absm = ((m < 0) ? -m : m);  
    if (absn == absm)  
        return absn;  
    else  
        return max(absn, absm);  
}
```

Test Coverage

line — every line reached

```
int max(int n, int m) {  
✓   if (n > m)  
✓       return n;  
✓   else  
✓       return m;  
}
```



```
int maxabs(int n, int m) {  
✓   int absn = ((n < 0) ? -n : n);  
✓   int absm = ((m < 0) ? -m : m);  
✓   if (absn == absm)  
✓       return absn;  
✓   else  
✓       return max(absn, absm);  
}
```

Test Coverage

statement/expression — each statement/expression reached

```
int max(int n, int m) {  
    if (n > m)  
        return n;  
    else  
        return m;  
}
```

```
int maxabs(int n, int m) {  
    int absn = ((n < 0) ? -n : n);  
    int absm = ((m < 0) ? -m : m);  
    if (absn == absm)  
        return absn;  
    else  
        return max(absn, absm);  
}
```

Test Coverage

branch — each branch of every conditional taken

```
int fact(int n) {  
    int x = 1;  
  
    do {  
        x = x * n;  
        n = n - 1;  
    } while (n > 0);  
  
    return x;  
}
```

Test Coverage

branch — each branch of every conditional taken

```
int fact(int n) {  
    int x = 1;  
  
    do {  
        x = x * n;  
        n = n - 1;  
    } while (n > 0);  
  
    return x;  
}
```

should try both **true** and **false**

Test Coverage

path — each control combination taken

```
int max_of_three(int n, int m, int p) {  
    int r = n;  
    if (m > r)  
        r = m;  
    if (p > r)  
        r = p;  
    return r;  
}
```

Test Coverage

path — each control combination taken

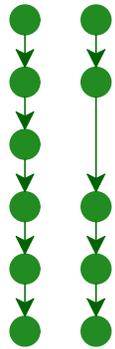
```
int max_of_three(int n, int m, int p) {  
    int r = n;  
    if (m > r)  
        r = m;  
    if (p > r)  
        r = p;  
    return r;  
}
```



Test Coverage

path — each control combination taken

```
int max_of_three(int n, int m, int p) {  
    int r = n;  
    if (m > r)  
        r = m;  
    if (p > r)  
        r = p;  
    return r;  
}
```

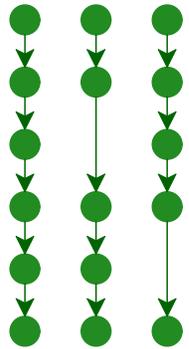


The control flow graph consists of two vertical paths of nodes connected by arrows. The left path has 6 nodes corresponding to the lines of code: the assignment of r to n, the if condition (m > r), the assignment of r to m, the if condition (p > r), the assignment of r to p, and the return statement. The right path has 2 nodes: the if condition (m > r) and the if condition (p > r). Arrows point downwards from each node to the next in both paths. A vertical arrow also connects the top node of the right path to the top node of the left path, indicating that the right path branches off from the left path at the first if statement.

Test Coverage

path — each control combination taken

```
int max_of_three(int n, int m, int p) {  
    int r = n;  
    if (m > r)  
        r = m;  
    if (p > r)  
        r = p;  
    return r;  
}
```

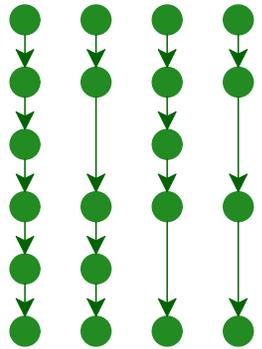


The diagram illustrates the control flow graph for the `max_of_three` function. It consists of three vertical paths of nodes, each connected by downward-pointing arrows. The first path represents the execution flow when `m > r` is false, starting from the assignment `r = n` and proceeding through the `if (p > r)` block to the `return r` statement. The second path represents the execution flow when `m > r` is true, starting from the `if (m > r)` block, then the assignment `r = m`, then the `if (p > r)` block, and finally the `return r` statement. The third path represents the execution flow when `m > r` is true and `p > r` is true, starting from the `if (m > r)` block, then the assignment `r = m`, then the `if (p > r)` block, and finally the `return r` statement. The nodes are represented by green circles, and the arrows indicate the direction of control flow.

Test Coverage

path — each control combination taken

```
int max_of_three(int n, int m, int p) {  
    int r = n;  
    if (m > r)  
        r = m;  
    if (p > r)  
        r = p;  
    return r;  
}
```

A control flow graph (CFG) for the function max_of_three. It consists of four vertical paths of nodes, each connected by downward-pointing arrows. The first path has 6 nodes, corresponding to the lines: int r = n; (green highlight), if (m > r) (green highlight), r = m; (green highlight), if (p > r) (green highlight), r = p; (green highlight), and return r; (green highlight). The second path has 2 nodes, corresponding to the if (m > r) condition and the r = m; assignment. The third path has 2 nodes, corresponding to the if (p > r) condition and the r = p; assignment. The fourth path has 1 node, corresponding to the return r; statement. The nodes in the second and third paths are connected to the first path at the if (m > r) and if (p > r) lines respectively. The nodes in the fourth path are connected to the first path at the return r; line.

Test Coverage

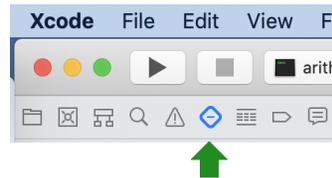
Even more:

- **condition** — each boolean true and false
- **value** — each [common] value at every possibility
- **edge** — each control transfer taken
- **modified condition/decision** — each boolean matters
- ...

Line or expression coverage is practical and useful

Testing in Xcode

Start by clicking here:



Then click “+” in the bottom left

Select “New Unit Test Target...”

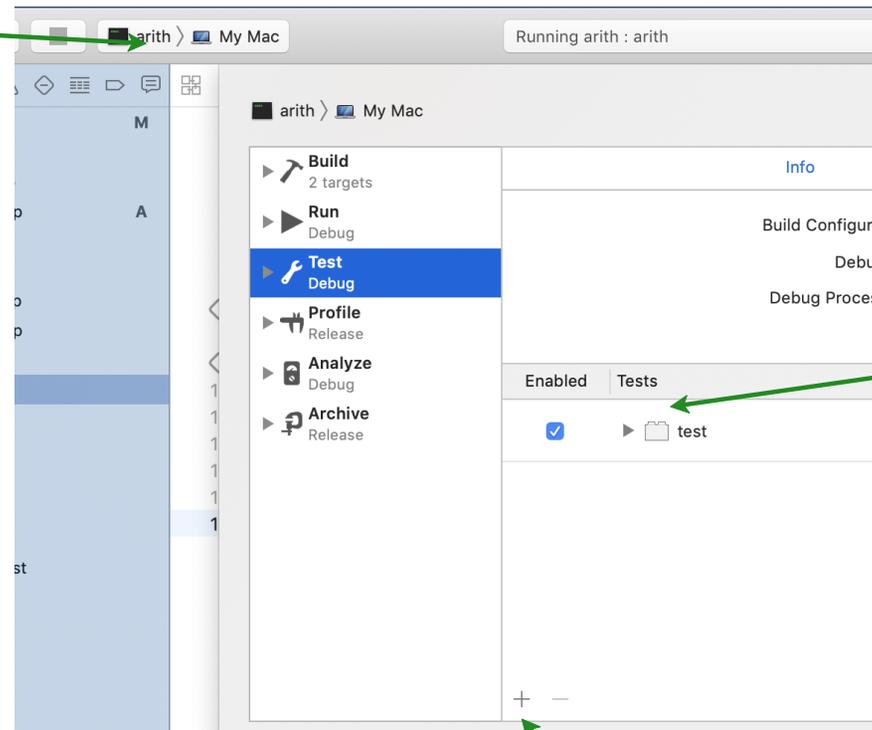
Pick “Objective-C” for the language

All of the project changes are part of the project that you probably have checked in to your Git repo

Testing in Xcode

Connect a test target to your main target:

1. Pick "Edit Scheme..."

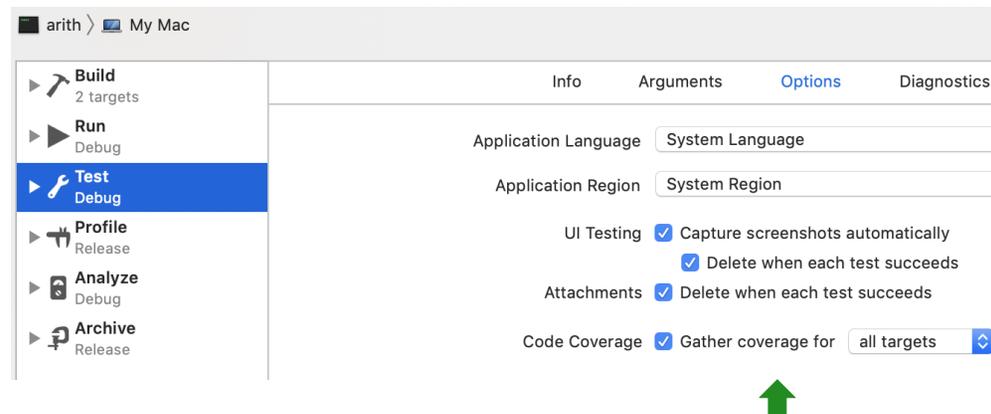


2. Click "+" and select test target

3. Should show up here

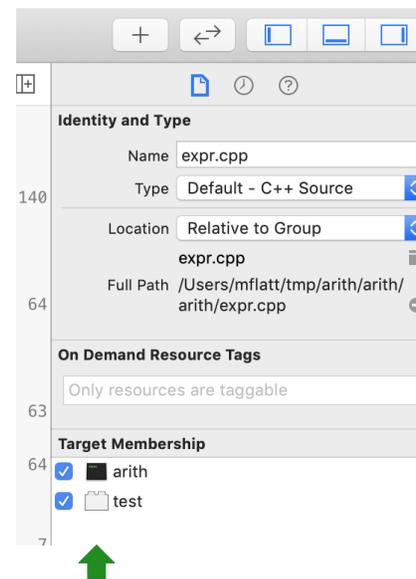
Testing in Xcode

Enable code coverage:



Testing in Xcode

For each non-main file, add to your new test target:



Testing in Xcode

Adjust created .m file:

```
#import <XCTest/XCTest.h>
#include "run.h"

@interface test : XCTestCase
@end

@implementation test
- (void)testAll {
    if (!run_tests())
        XCTFail(@"failed");
}
@end
```

Testing in Xcode

Add glue code in new file `run.h`:

```
extern bool run_tests(void);
```

Testing in Xcode

Add glue code in new file `run.cpp`:

```
extern "C" {
#include "run.h"
};

#define CATCH_CONFIG_RUNNER
#include "../catch.h"

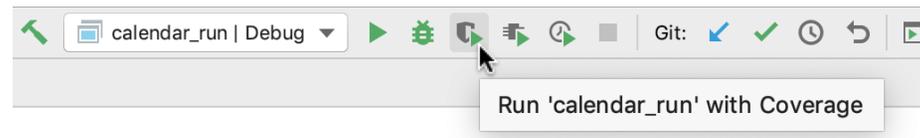
bool run_tests() {
    const char *argv[] = { "arith" };
    return (Catch::Session().run(1, argv) == 0);
}
```

Testing in Xcode

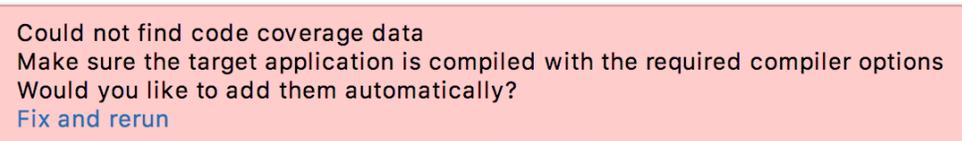
- Use **Test** ⌘U instead of **Run** ⌘R from the **Project** menu
- Turn on **Code Coverage** in the **Editor** menu
- Look for pink bars along the right edge of your code ⇒ uncovered

Testing in CLion

To run with coverage:



First run:



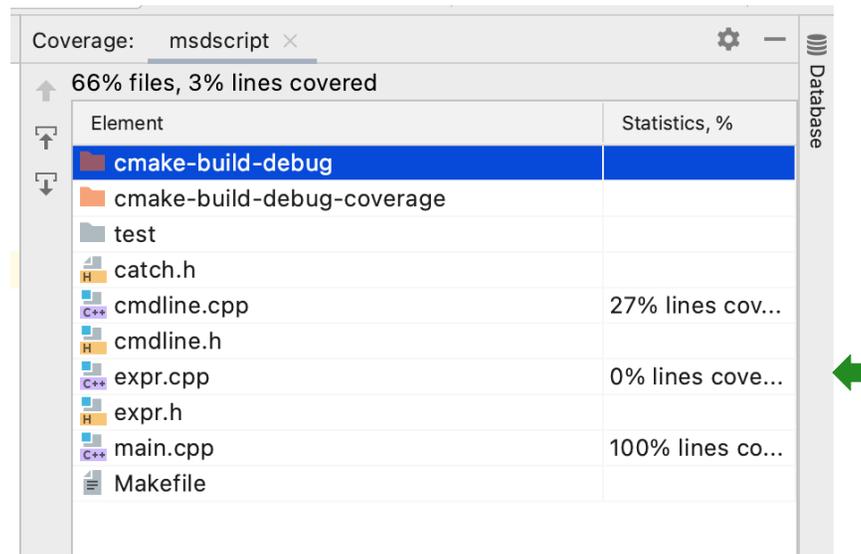
Click **Fix and rerun**

<https://www.jetbrains.com/help/clion/code-coverage-clion.html>

Beware: some changes will affect only your project workspace,
which you probably exclude from your Git repo

Testing in CLion

When you run with coverage (again), probably the interesting file has 0% coverage:



Coverage: msdscrip ×

66% files, 3% lines covered

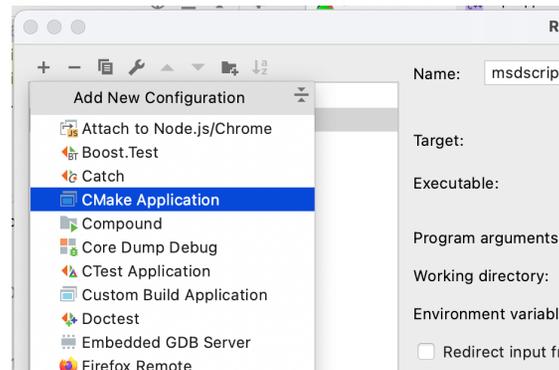
Element	Statistics, %
cmake-build-debug	
cmake-build-debug-coverage	
test	
catch.h	
cmdline.cpp	27% lines cov...
cmdline.h	
expr.cpp	0% lines cove...
expr.h	
main.cpp	100% lines co...
Makefile	

That's because no tests were run

Testing in CLion

Add `--test` when running with coverage:

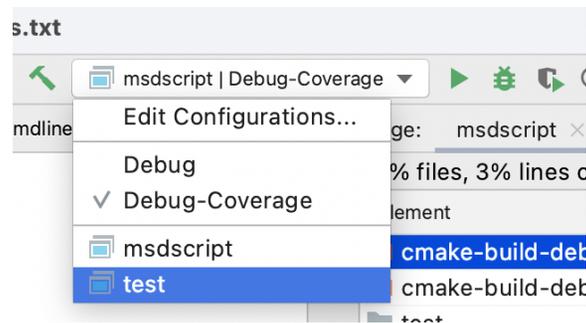
- Go to **Run** → **Edit Configurations...**
- Click **+** and add a new **CMake Application**



- Name it something like **test**
- Set the **Program arguments:** field to `--test`

Testing in CLion

- Pick the **test** configuration while keeping **Debug-Coverage** still checked



- Run with coverage again, and since your program runs the test suite when `--test` is the argument, now you get usefule coverage
- Look for pink bars along the editor left edge to find uncovered lines

Makefile and Testing

Makefile idea: create a `test` “phony” target:

```
....  
  
msdscript: $(OBJS)  
           c++ -o msdscript $(OBJS)  
  
.PHONY: test  
test: msdscript  
      ./msdscript --test  
  
....
```

Then, `make test` builds and runs tests

GitHub Actions

You should always runs your tests, but computers are good at remembering things that people forget

On Github: **Actions** → **set up a workflow yourself**

Use this text:

```
name: CI

on: [push]

jobs:
  build:

    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v1
      - name: Build and run tests
        run: make test
```

GitHub Actions

You should always runs your tests, but computers are good at remembering things that people forget

On Github: **Actions** → **set up a workflow yourself**

Use this text:

```
name: CI

on: [push]

jobs:
  build:

    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v1
      - name: Build and run tests
        run: make test
```

or add to repo as
`.github/workflows/main.yml`

GitHub Actions

You should always runs your tests, but computers are good at remembering things that people forget

On Github: **Actions** → **set up a workflow yourself**

If your **Makefile** is in `path/to/dir` within the repo:

```
name: CI

on: [push]

jobs:
  build:

    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v1
      - name: Build and run tests
        run: make test
        working-directory: path/to/dir
```

GitHub Actions

You should always runs your tests, but computers are good at remembering things that people forget

On Github: **Actions** → **set up a workflow yourself**

If your **Makefile** is in `path/to/dir` within the repo:

```
name: CI

on: [push]

jobs:
  build:

    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v1
      - name: Build and run tests
        run: make test
        working-directory: path/to/dir
```

Don't include starting slash or the name of your repo directory

Part 2: Local Binding

Variable Binding

In homework, you added variables to MSDscript

But we don't yet have a way to give a variable a value

Next: add a declaration form called `_let`

In general, MSDscript keywords will start with an underscore

Let Binding

```
_let x = 5  
_in  x + 1
```

Let Binding

```
_let x = 5  
_in x + 1
```

Result is 6

Let Binding

```
_let x = 5  
_in x + 1
```

Similar to

```
{  
  int x = 5;  
  x + 1;  
}
```

because `x` is visible only in the `_in` part...

Let Binding

```
_let x = 5  
_in x + 1
```

... but just *one* expression must be after `_in...`

Let Binding

```
_let x = 5  
_in  x + 1
```

... and the whole thing is still an expression

Let Binding

```
_let x = 5  
_in x + 1
```

```
(_let x = 5  
_in x + 1) * 2
```

Let Binding

```
_let x = 5  
_in x + 1
```

```
(_let x = 5  
_in x + 1) * 2
```

result is 12

Let Binding

```
_let x = 5  
_in x + 1
```

```
(_let x = 5  
_in x + 1) * 2
```

```
_let x = 5  
_in x + 1 * 2
```

Let Binding

```
_let x = 5  
_in x + 1
```

```
(_let x = 5  
_in x + 1) * 2
```

```
_let x = 5  
_in x + 1 * 2
```

result is 7

Let Grammar

$\langle \text{expr} \rangle = \langle \text{number} \rangle$
| $\langle \text{expr} \rangle + \langle \text{expr} \rangle$
| $\langle \text{expr} \rangle * \langle \text{expr} \rangle$
| $\langle \text{variable} \rangle$
| $_let \langle \text{variable} \rangle = \langle \text{expr} \rangle _in \langle \text{expr} \rangle$

right-hand side (RHS) body

Interpreting Let

```
_let x = 5  
_in x + 1
```

⇒

5 + 1

⇒

6

Interpreting `_let` can use `subst`

```
(new Add(new Var("x"), new Num(1)))  
->subst("x", new Num(5))  
->interp()
```

Interpreting Let

```
_let x = 5 + 2 => 7  
_in x + 1
```

⇒

7 + 1

⇒

8

Interpret binding RHS before subst

```
int n = (new Add(new Num(5), new Num(2)))  
->interp();  
new Num(n);
```

Nested Let Binding: Body

```
_let x = 5
_in  _let x = 6
      _in  x + 1
```

Nested Let Binding: Body

```
_let x = 5  
_in  _let x = 6  
      _in  x + 1
```

means 6, not 5

Nested Let Binding: Body

```
_let x = 5  
_in _let x = 6  
    _in x + 1
```

Analogous to

```
{  
  int x = 5;  
  {  
    int x = 6;  
    x + 1;  
  }  
}
```

Nested Let Binding: Body

```
_let x = 5  
_in  _let x = 6  
      _in  x + 1
```

Substitute **x** with **5** in

```
_let x = 6  
_in  x + 1
```

 should not change

```
_let x = 6  
_in  x + 1
```

Nested Let Binding: Body

```
_let x = 5
_in  _let x = 6
     _in  x + 1
```

Substitute **x** with **5** in

```
_in x + 1
```

 should change to

```
_in 5 + 1
```

Substitution of **<variable>** with **<expr>** at **_let**:

- bind *same* **<variable>**: don't substitute in the *body*
- bind *different* **<variable>**: substitute in the *body*

Nested Let Binding: Body

```
_let x = 5  
_in  _let x = 6  
      _in x + 1
```

In other words, substitution replaces **free variables**, and it does not replace **bound variables**

x is **bound** in

```
_let x = 5  
_in x + 1
```

Nested Let Binding: Body

```
_let x = 5  
_in  _let x = 6  
      _in x + 1
```

In other words, substitution replaces **free variables**, and it does not replace **bound variables**

x is **free** in

```
x + 1
```

Nested Let Binding: Body

```
_let x = 5
_in  _let x = 6
     _in  x + 1
```

In other words, substitution replaces **free variables**, and it does not replace **bound variables**

x is **free** in

```
_let z = 5
_in x + 1
```

Nested Let Binding: RHS

```
_let x = 5  
_in  _let x = x + 2  
      _in  x + 1
```

Substitution of `<variable>` with `<expr>` at `_let:`

- Always substitute in the *right-hand side*