Part 1: 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

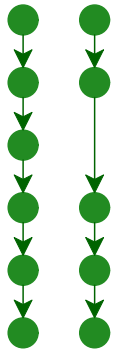
path — each control combination taken

```
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

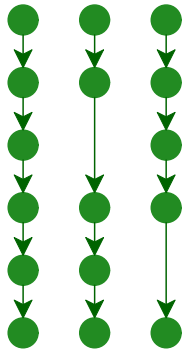
```
int max_of_three(int n, int m, int p) {
  int r = n;
  if (m > r)
    r = m;
  if (p > r)
    r = p;
  return r;
}
```



Test Coverage

path — each control combination taken

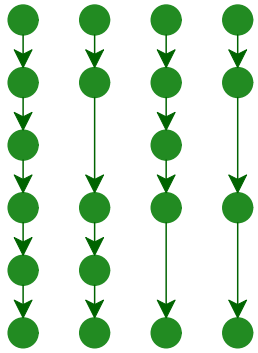
```
int max_of_three(int n, int m, int p) {
  int r = n;
  if (m > r)
    r = m;
  if (p > r)
    r = p;
  return r;
}
```



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

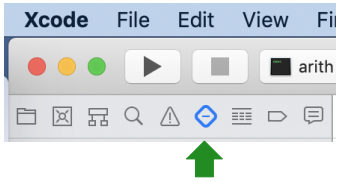
Even more:

- **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..."

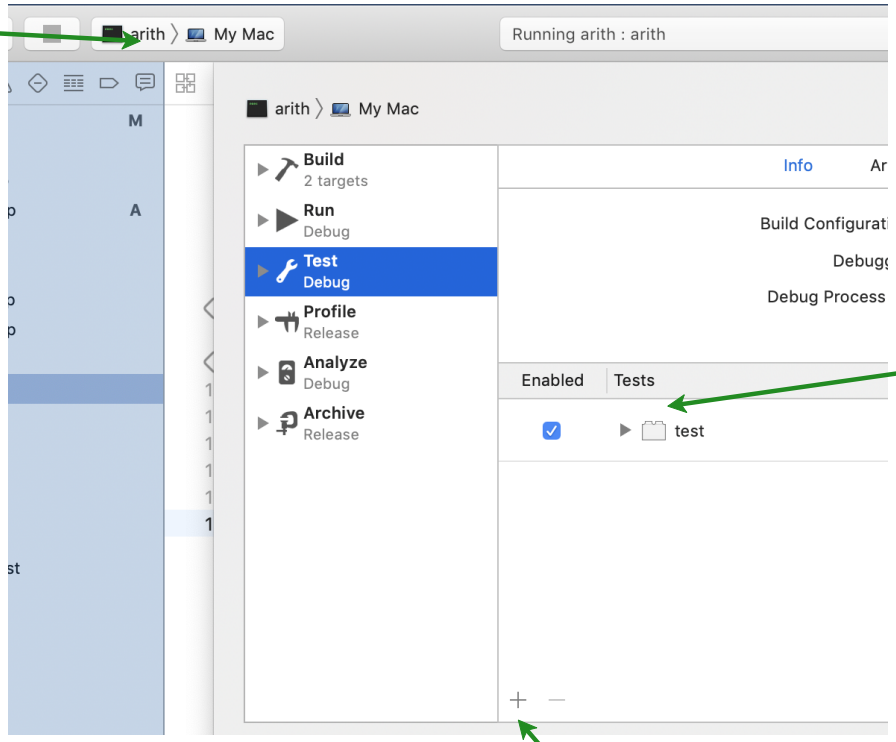
Pick "Objective-C" for the language

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..."

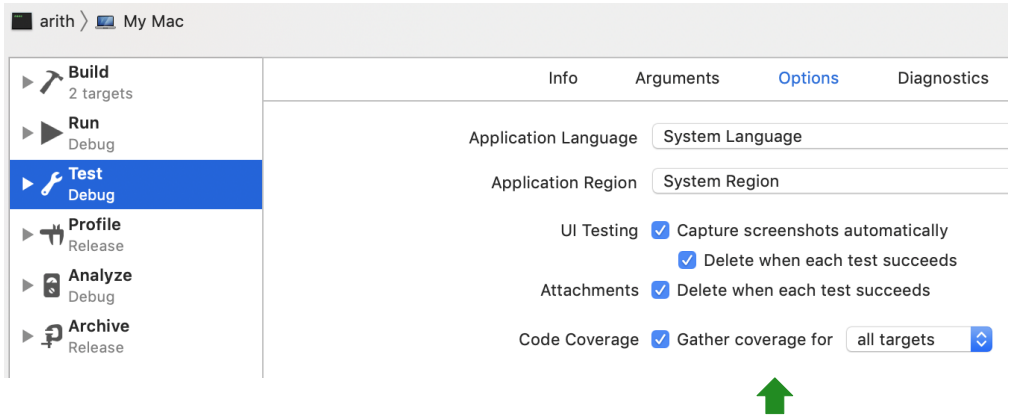


3. Should show up here

2. Click ``+" and select test target

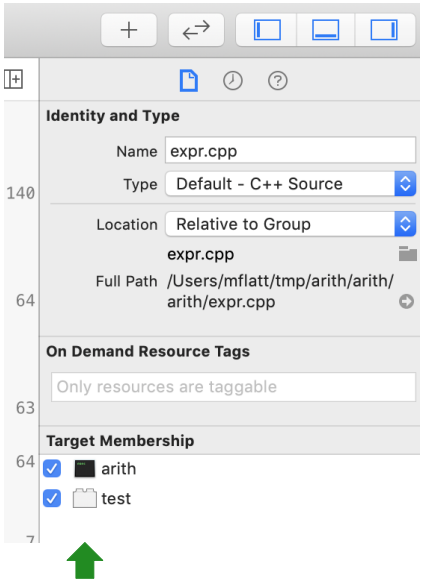
Testing in Xcode

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`**:

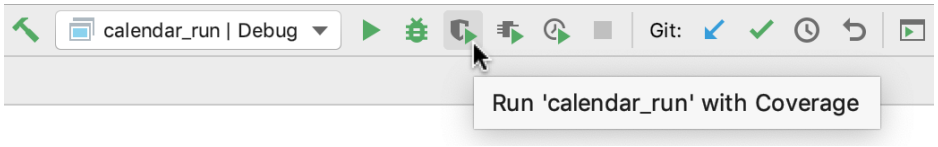
```
extern "C" {
#include "run.h"
};

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

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

Testing in Xcode

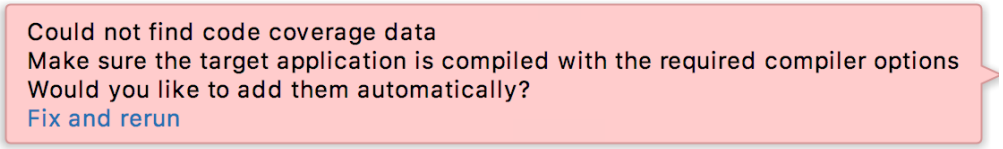
- Use **Test** ⌘U instead of **Run** ⌘R from the **Project** menu
- Turn on **Code Coverage** in the **Editor** menu
- Look for pink bars along the right edge of your code ⇒ uncovered

Testing in CLion

To run with coverage:



First run:



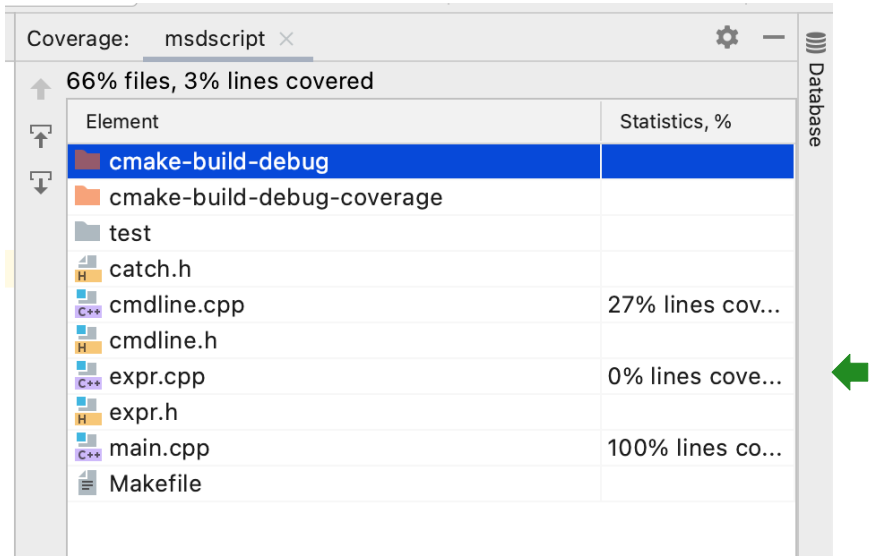
Click **Fix and rerun**

`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:

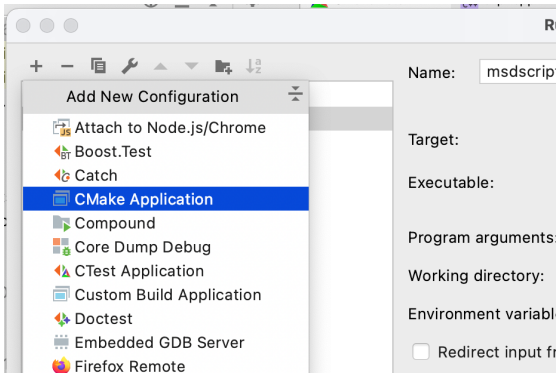


That's because no tests were run

Testing in CLion

Add `--test` when running with coverage:

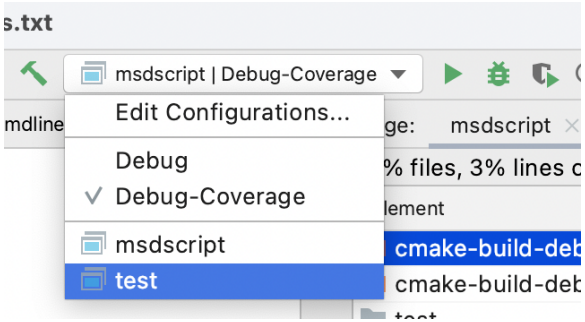
- Go to **Run** → **Edit Configurations...**
- Click **+** and add a new **CMake Application**



- 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:

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

```
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**

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

⟨expr⟩ = ⟨number⟩
| ⟨expr⟩ `+` ⟨expr⟩
| ⟨expr⟩ `*` ⟨expr⟩
| ⟨variable⟩
| `_let` ⟨variable⟩ `=` ⟨expr⟩ `_in` ⟨expr⟩

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
_in  x + 1
```

⇒ 7

⇒

```
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 `_let y = 6` `_in x + 1` should change to `_let y = 6` `_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*