CS 6015: Software Engineering

Spring 2024

Lecture 8: Code Coverage

This Week

- Defensive programming
- Testing / Code Coverage
- Homework 4

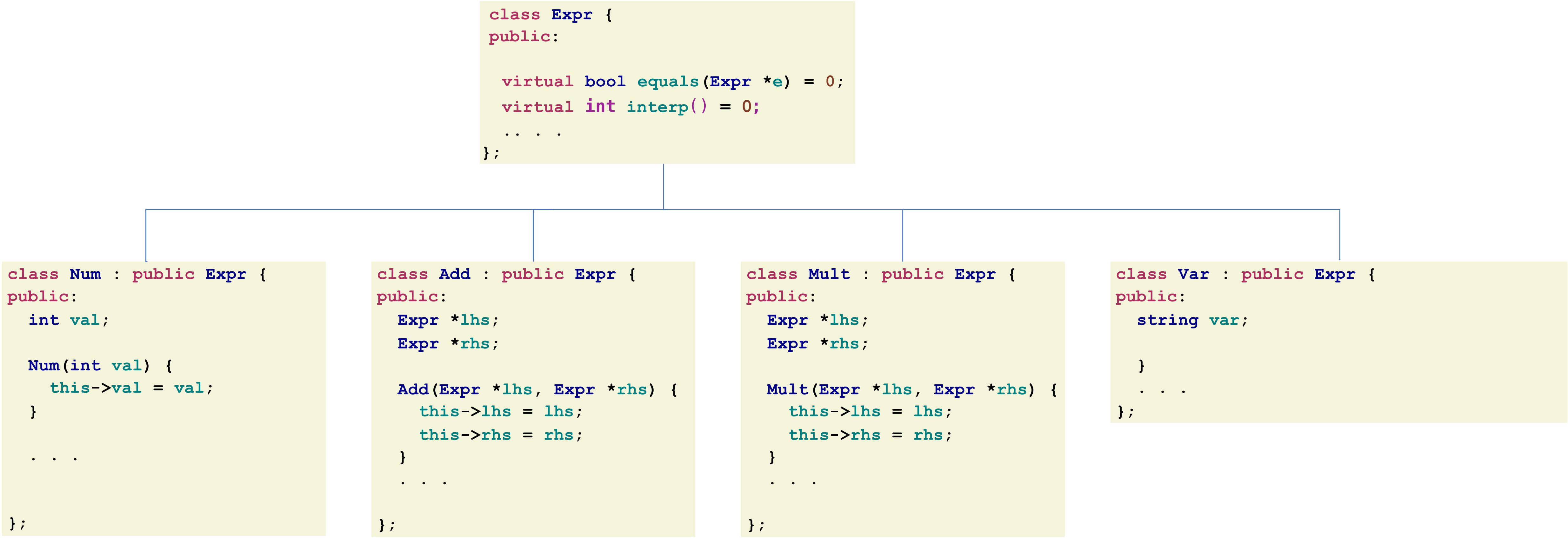
Next Week

- Documentation
- Let Binding (Project related)
- Parsing (Project related)

Project Related - MSDscript

- Two main parts:
  - Representation Classes (What we’ve been doing so far)
  - Parsing

MSDscript: Representation Classes / methods



```
class Expr {
public:

  virtual bool equals(Expr *e) = 0;
  virtual int interp() = 0;
  .. . .
};
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }

  . . .


};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
  . . .

};
```

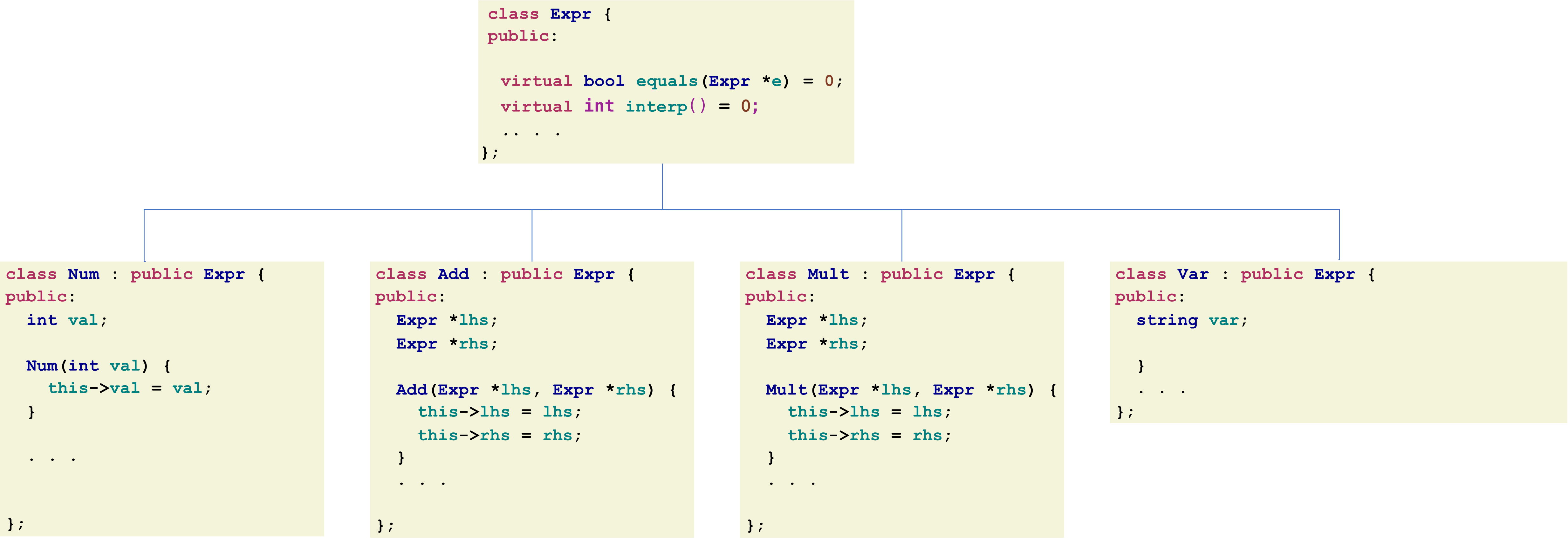
```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
  . . .

};
```

```
class Var : public Expr {
public:
  string var;

  }
  . . .
};
```

MSDscript: Grammar

```
⟨expr⟩    =  ⟨number⟩
          |  ⟨expr⟩ + ⟨expr⟩
          |  ⟨expr⟩ * ⟨expr⟩
          |  ⟨variable⟩
          | _let  ⟨variable⟩ = ⟨expr⟩  _in  ⟨expr⟩ Coming soon
          | …. More (later)
```

MSDscript: Representation Classes / methods



```
class Expr {
public:

  virtual bool equals(Expr *e) = 0;
  virtual int interp() = 0;
  .. . .
};
```

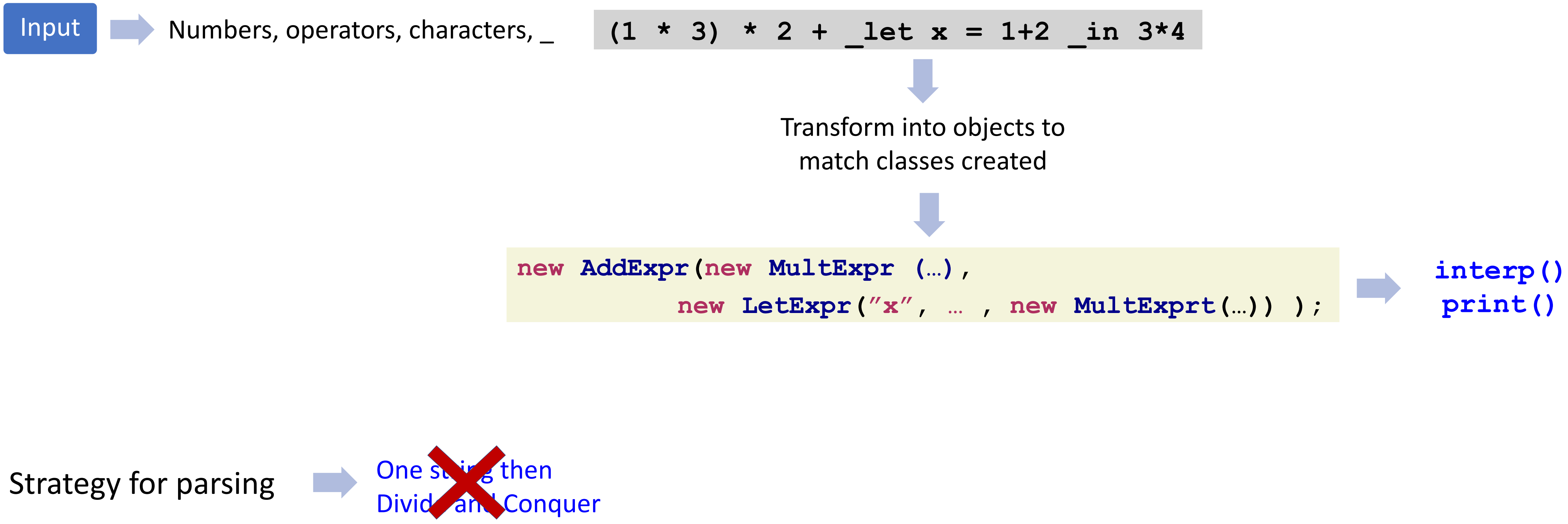
```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }

  . . .


};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
  . . .

};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
  . . .

};
```

```
class Var : public Expr {
public:
  string var;

  }
  . . .
};
```

MSDscript: Parsing



Input → Numbers, operators, characters, _

```
(1 * 3) * 2 + _let x = 1+2 _in 3*4
```

Transform into objects to match classes created

```
new AddExpr(new MultExpr (…),
            new LetExpr(”x”, … , new MultExprt(…)) );
```

```
interp()
print()
```

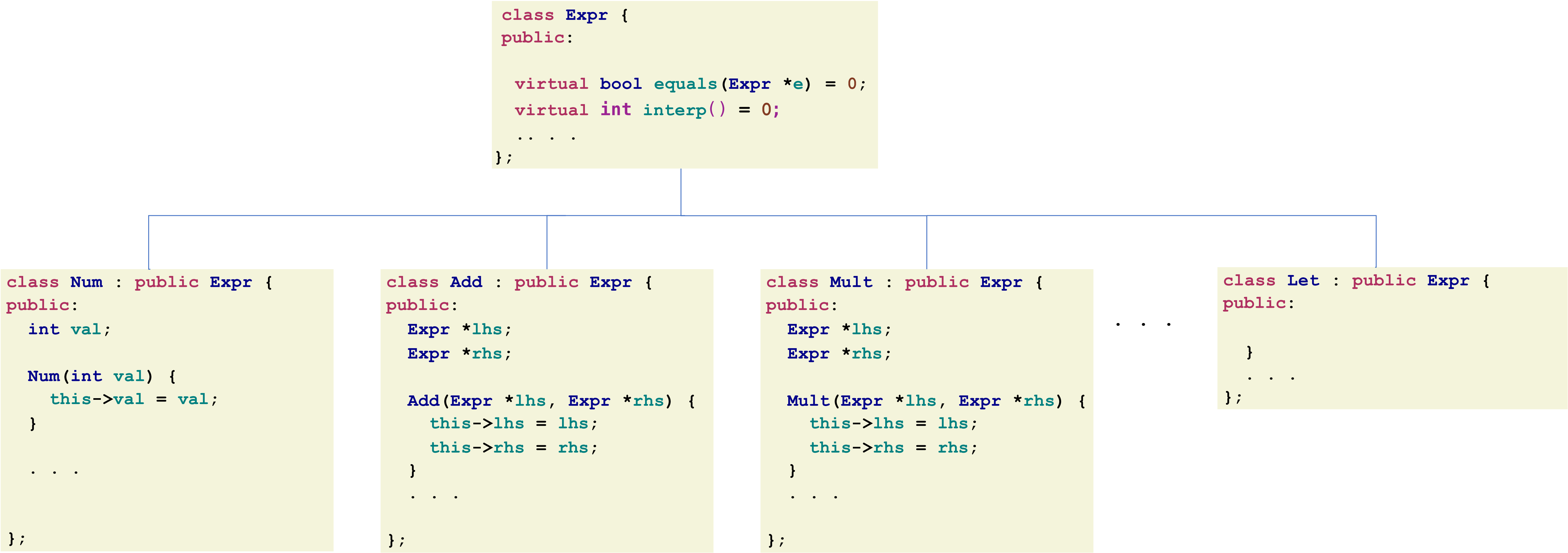
Strategy for parsing → ~~One string then Divide and Conquer~~

Stream of characters

Next Week

- Extend the grammar to include **Let**
- Add class representation for the new grammar
- Parsing

MSDscript: Representation Classes / methods



```
class Expr {
public:

  virtual bool equals(Expr *e) = 0;
  virtual int interp() = 0;
  .. . .
};
```

```
class Num : public Expr {
public:
  int val;

  Num(int val) {
    this->val = val;
  }

  . . .


};
```

```
class Add : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Add(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
  . . .

};
```

```
class Mult : public Expr {
public:
  Expr *lhs;
  Expr *rhs;

  Mult(Expr *lhs, Expr *rhs) {
    this->lhs = lhs;
    this->rhs = rhs;
  }
  . . .

};
```

. . .

```
class Let : public Expr {
public:

   }
   . . .
};
```

Plan

- Recall: Testing
- Code coverage
- Coverage types
- Code coverage in Xcode
- Continuous integration
    - GitHub actions

Testing

- Our team leader / supervisor asked whether the test cases are sufficient.
- How to make sure that the tests cover all possible cases?
- How do you know whether a program is tested well?
- Recall from testing:
  - Random testing
    - Not sufficient
  - Exhaustive testing
    - Hard to achieve / time consuming
- Solution??

Code coverage

- Describes how much of your code is executed while testing.
- Expression of the goodness of your test cases.
- Many metrics/notions are used:
  - Statement/Line coverage
  - Branch coverage
  - Path coverage
  - Function coverage
  - Loop coverage

Statement/Line Coverage

```
int returnInput(int input, bool cond1, bool cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

`Statement Coverage for CHECK( returnInput(2, true, true, true) == 2 ); ?`

`Statement Coverage for CHECK( returnInput(x, true, true, true) == x ); ?`

Statement/Line Coverage

- Refers to the percentage of statements in the code that have been executed by the test cases

```
int returnInput(int input, bool cond1, bool cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

```
Statement Coverage for CHECK( returnInput(2, true, true, true) == 2 );
Statement Coverage for CHECK( returnInput(x, true, true, true) == x );
```

100%

Branch Coverage

```
int returnInput(int input, bool cond1, bool cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

```
Branch Coverage for CHECK( returnInput(2, true, true, true) == 2 ); ?

Branch Coverage for CHECK( returnInput(x, true, true, true) == x ); ?
```

Branch Coverage

- Refers to the percentage of branches that have been executed; Each possible branch counted separately

```
int returnInput(int input, bool cond1, bool cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

```
Branch Coverage for CHECK( returnInput(2, true, true, true) == 2 );
Branch Coverage for CHECK( returnInput(x, true, true, true) == x );
```

50%

Branch Coverage

- Refers to the percentage of branches that have been executed; Each possible branch counted separately

- An “if” statement has 2 branches (even if there is not “else” statement):
  - a branch that executes when the condition is true, and
  - a branch that executes when the condition is false

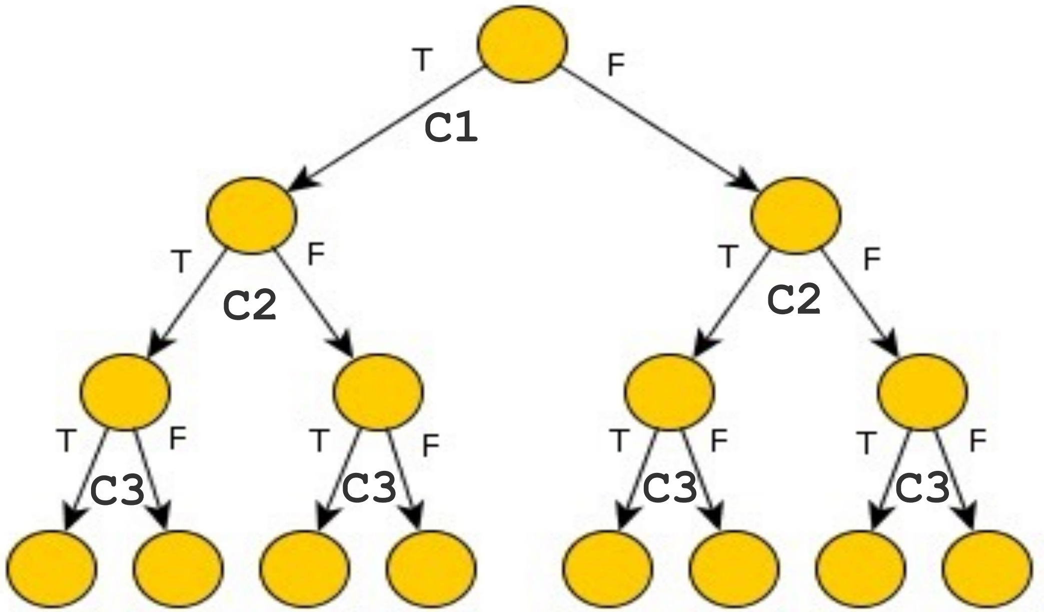
- A “switch” can have many branches

Path Coverage

```
int returnInput(int input, bool cond1, bool cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

Path Coverage for `CHECK( returnInput(AnyNum, true, true, true) == AnyNum ); ?`

Path Coverage



```
int returnInput(int input, bool cond1,
bool cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

`Path Coverage for CHECK( returnInput(AnyNum, true, true, true) == AnyNum );`

1/8 path coverage

Function Coverage

```
int returnInput(int input, bool cond1, bool
cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

```
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);
}
```

```
Function Coverage for:
CHECK( returnInput(x, true, true, true) == x );
CHECK( max(100, 50) == 100 );
```

2/3 functions called

Function Coverage

✓

```
int returnInput(int input, bool cond1, bool
cond2, bool cond3){

    int x = input;
    int y = 0;

    if (cond1)
      x++;
    if (cond2)
      x--;
    if (cond3)
      y=x;

    return y;
}
```

✓

```
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);
}
```

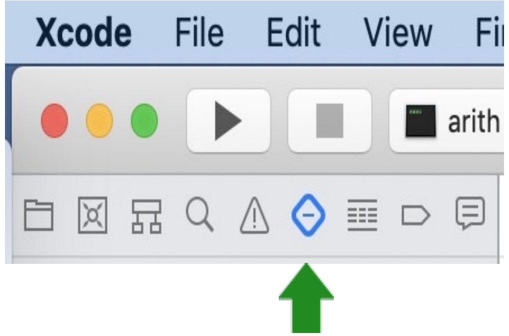
100% Function Coverage when every function is called

Code coverage vs Test coverage

- Code coverage
  - Verifies the extent to which the code has been executed
  - Levels: Line/branch/path/function/….
- Test coverage
  - Measures how much of the feature set is covered
  - Types: Feature/Risk/Requirements

Testing in Xcode

Start by clicking here:



Then click "+" in the bottom left Select "New Unit Test Target..."

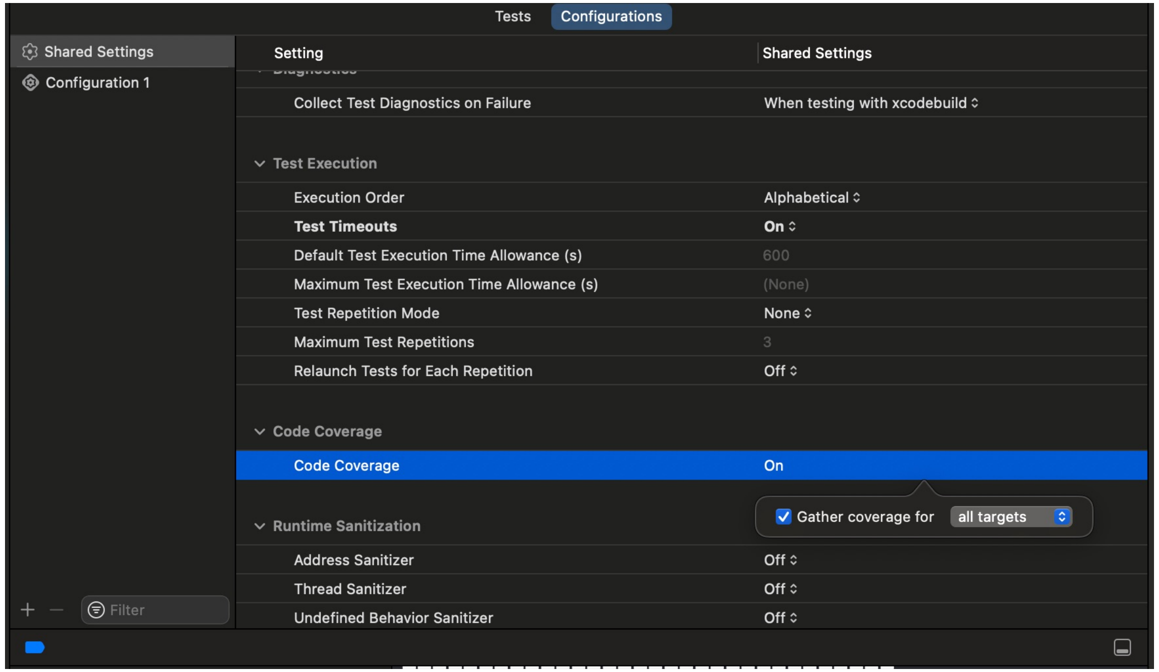
If it shows Missing Test Plan, then you need to create new Test Plan

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

Enable code coverage (Found under Configuration of your Test Plan):

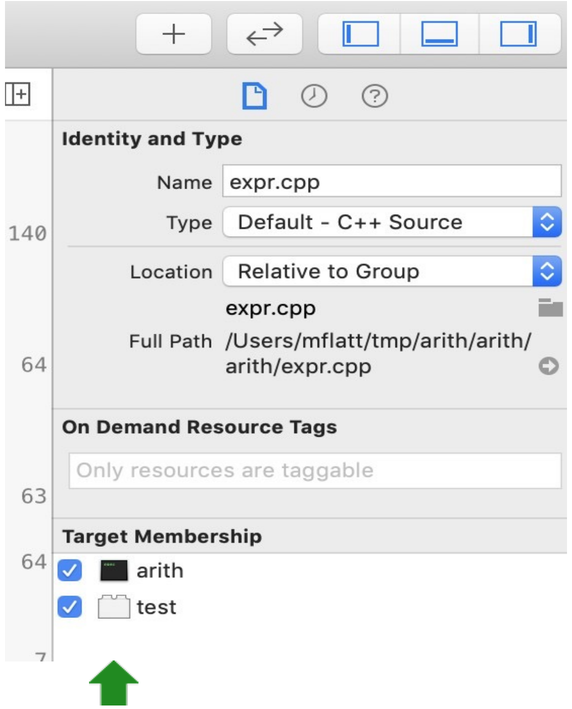


Setting	Shared Settings
Diagnostics	
Collect Test Diagnostics on Failure	When testing with xcodebuild ◇
Test Execution	
Execution Order	Alphabetical ◇
Test Timeouts	**On ◇**
Default Test Execution Time Allowance (s)	600
Maximum Test Execution Time Allowance (s)	(None)
Test Repetition Mode	None ◇
Maximum Test Repetitions	3
Relaunch Tests for Each Repetition	Off ◇
Code Coverage	
Code Coverage	On
	☑ Gather coverage for all targets ◇
Runtime Sanitization	
Address Sanitizer	Off ◇
Thread Sanitizer	Off ◇
Undefined Behavior Sanitizer	Off ◇

Testing in Xcode

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

- click on the file and check the Target Membership
- or right click then show file inspector



Identity and Type

Name: expr.cpp

Type: Default - C++ Source

Location: Relative to Group

expr.cpp

Full Path: /Users/mflatt/tmp/arith/arith/arith/expr.cpp

On Demand Resource Tags

Only resources are taggable

Target Membership

☑ arith

☑ test

Testing in Xcode

Adjust created **.m** fle:

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

@interface test : XCTestCase
@end

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

Testing in Xcode

Add glue code in new file **run.hpp**:

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

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

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

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

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

Testing in Xcode

- Use **Test** ⌘U instead of **Run**⌘R from the **Project** menu
- or switch to Test Plan, then right click and run testAll
- Turn on **Code Coverage** in the **Editor** menu
- Look for pink bars along the right edge of your code ⇒ uncovered

Continuous Integration

Continuous Integration (CI)

Continuous integration is the practice of running the steps that were traditionally performed during “integration” little and often throughout the development process, rather than waiting until code is complete before bringing it all together and testing it.

Ref: https://www.jetbrains.com/teamcity/ci-cd-guide/continuous-integration-vs-delivery-vs-deployment/

GitHub Actions

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

- On Github: **Actions → set up a workfow yourself**

- Use this text:

```
name: MSD
on:
  push:
jobs:
  build:
    runs-on: macos-latest
    steps:
     - name: Checkout v1
       uses: actions/checkout@v1
    - name: Run test
       run: make test
```

GitHub Actions

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

- On Github: **Actions → set up a workfow yourself**

- Use this text:

```
name: MSD
on:
  push:
jobs:
  build:
    runs-on: macos-latest
    steps:
     - name: Checkout v1
       uses: actions/checkout@v1
    - name: Run test
       run: make test
       working-directory: path/to/dir
```

If your **source files** and the **Makefile** is in **path/to/dir** within the repo, add:

working-directory: path/to/dir

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

GitHub Actions more

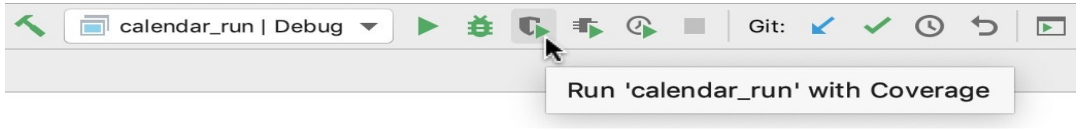
```
name: MSD
on:
  push:
    branches:
      - main
jobs:
  build:
    runs-on: macos-latest
    strategy:
      matrix: { dir: ['lab3', 'homework4'] }
    steps:
      - name: Checkout v1
        uses: actions/checkout@v1
      - name: Run test
        run: make test
        working-directory: ${{ matrix.dir }}
```

Adding GitHub actions for each folder

Testing in CLion

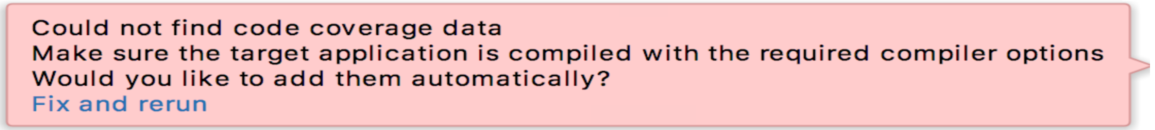
Testing in CLion

To run with coverage:



Run 'calendar_run' with Coverage

First run:



Could not find code coverage data
Make sure the target application is compiled with the required compiler options
Would you like to add them automatically?
Fix and rerun

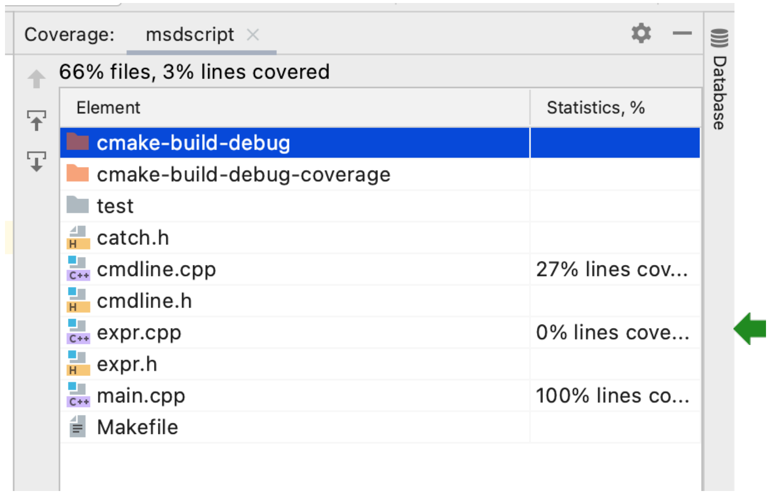
Click **Fix and rerun**

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 fle has 0% coverage:



66% files, 3% lines covered

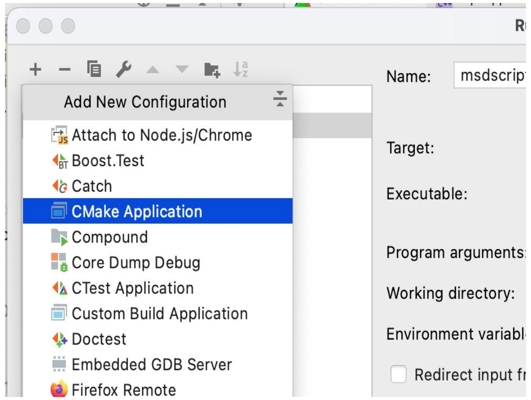
Element	Statistics, %
cmake-build-debug	
cmake-build-debug-coverage	
test	
catch.h	
cmdline.cpp	27% lines cov...
cmdline.h	
expr.cpp	0% lines cove...
expr.h	
main.cpp	100% lines co...
Makefile	

That’s because no tests were run

Testing in CLion

Add **--test** when running with coverage:

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

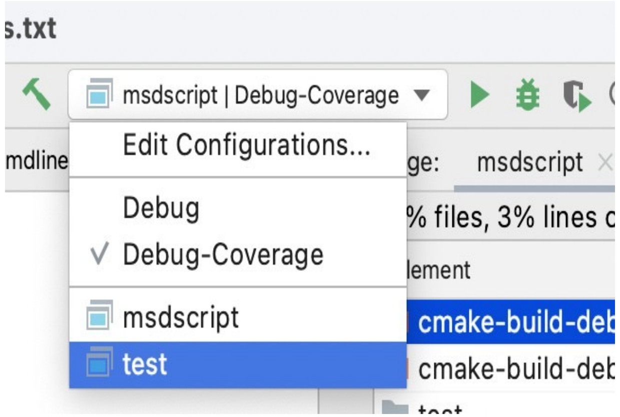


Name: msdscrip

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

Testing in CLion

- Pick the **test** confguration 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 fnd uncovered lines