# pgTAP Best Practices

David E. Wheeler

Kineticode, Inc.
PostgreSQL Experts, Inc.

PostgreSQL Conference West 2009

# OMG TAP WTF?

Test Anything Protocol (TAP) is a general purpose format for transmitting the result of test programs to a thing which interprets and takes action on those results. Though it is language agnostic, it is primarily used by Perl modules.

—Wikipedia

# What is TAP?

# What is TAP?

- What does that mean in practice?

# What is TAP?

- What does that mean in practice?
- Test output easy to interpret

# What is TAP?

- What does that mean in practice?
- Test output easy to interpret
- By humans

# What is TAP?

- What does that mean in practice?
- Test output easy to interpret
  - By humans
  - By computers

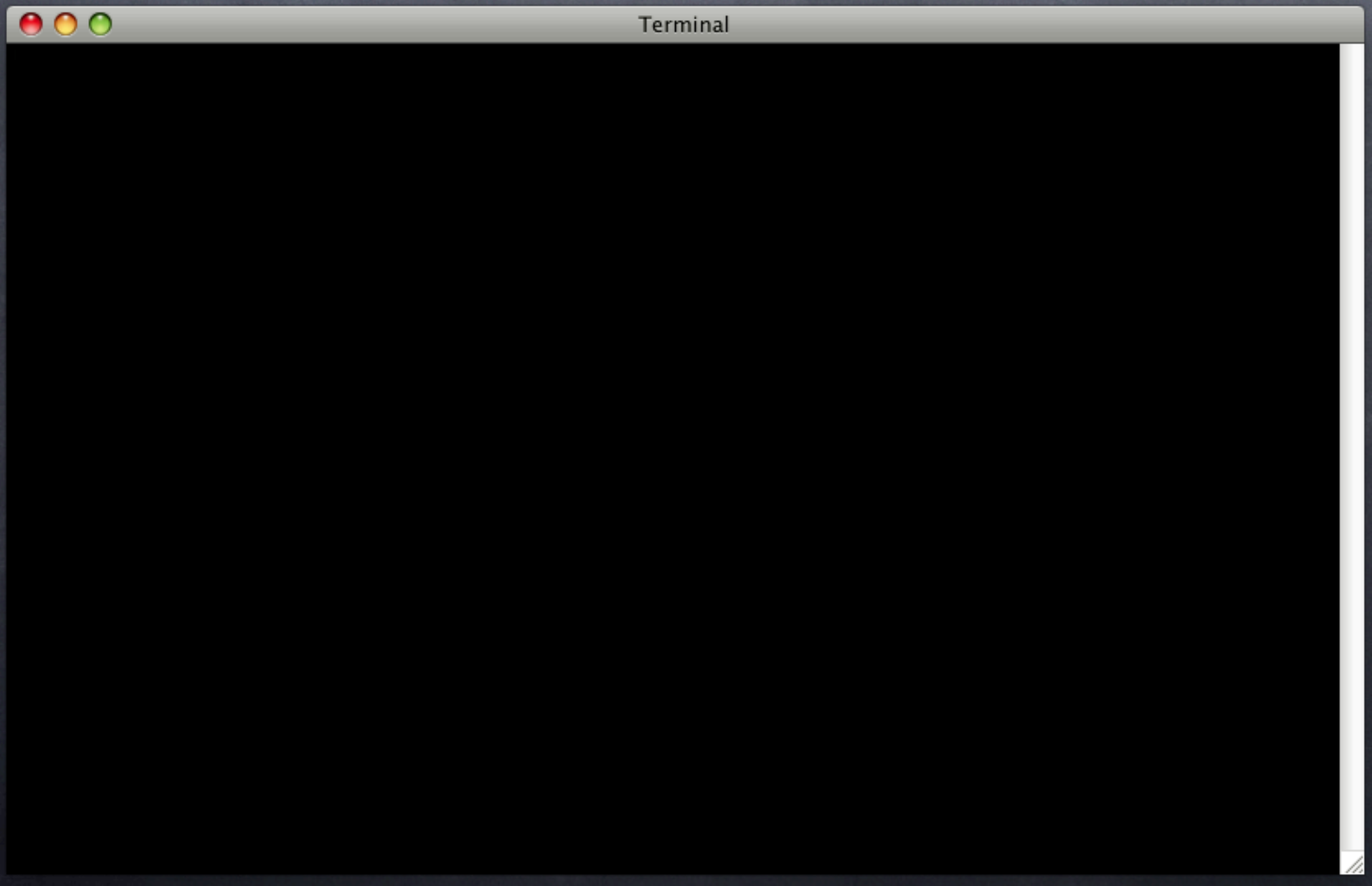
# What is TAP?

- What does that mean in practice?
- Test output easy to interpret
  - By humans
  - By computers
  - By aliens

# What is TAP?

- What does that mean in practice?
- Test output easy to interpret
  - By humans
  - By computers
  - By aliens
  - By gum!

# TAP Output



# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output

Terminal

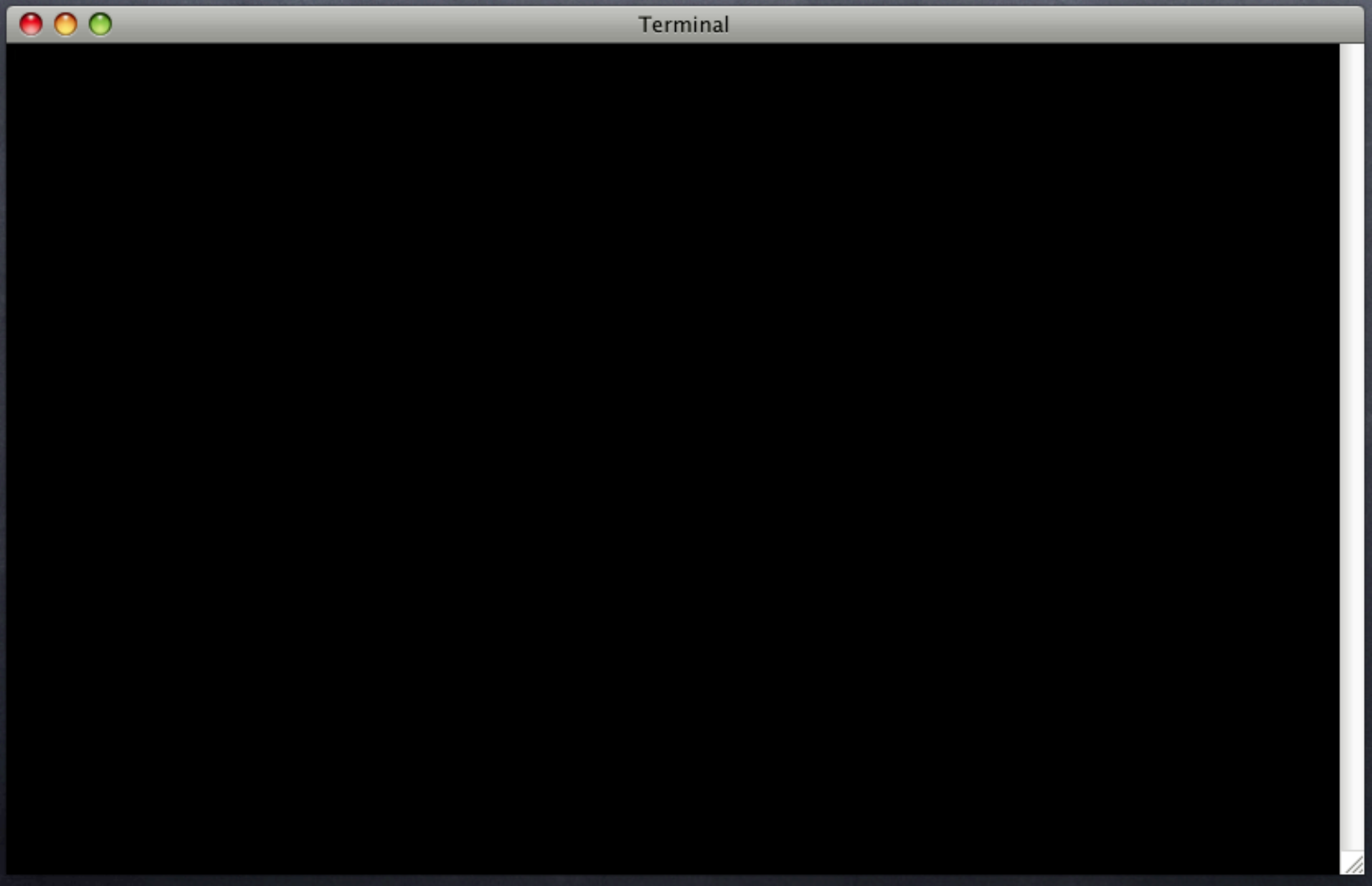
```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
not ok 4 - Should have a state
#   Failed test 'Should have a state'
#   at t/try.t line 12.
ok 5 - Should be able to reset
# Looks like you failed 1 test of 5.
```

# TAP Output



# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
ok 4 - Should have a state
ok 5 - Should be able to reset
```

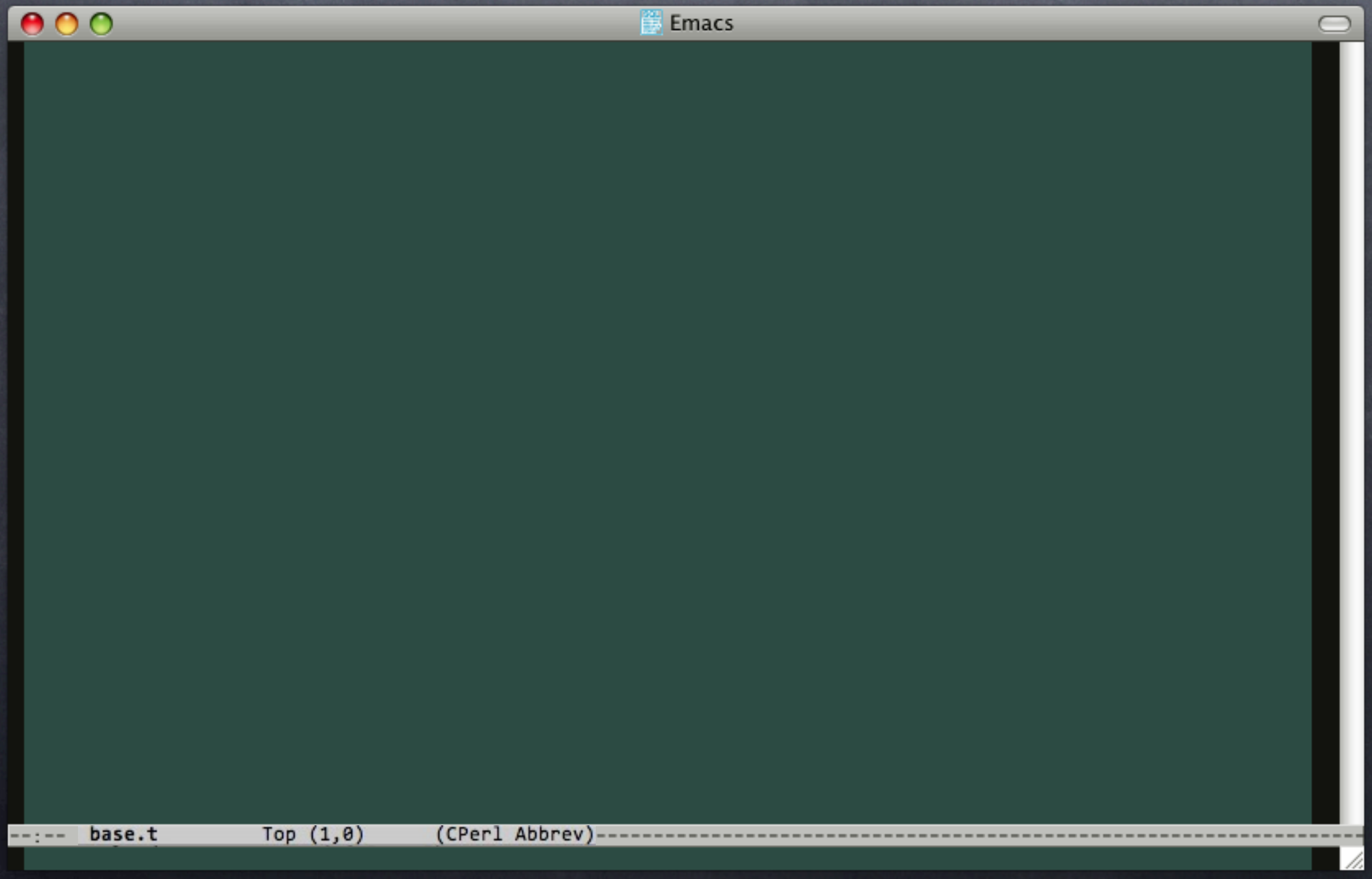
# TAP Output

Terminal

```
% perl -Ilib t/try.t
1..5
ok 1 - use FSA::Rules;
ok 2 - Create FSA::Rules object
ok 3 - Start the machine
ok 4 - Should have a state
ok 5 - Should be able to reset
```

## Think they passed?

# Writing Tests



# Writing Tests

Emacs

```
use Test::More tests => 5;

use_ok 'FSA::Rules';
ok my $r = FSA::Rules->new,
   'Create FSA::Rules object';
ok $r->start, 'Start the machine';
ok $r->curr_state, 'Should have a state';
ok $r->reset, 'Should be able to reset';
```

--:-- base.t Top (1,0) (CPerl Abbrev)----------

# Writing Tests

Emacs

```
use Test::More tests => 5;

use_ok 'FSA::Rules';
ok my $r = FSA::Rules->new,
   'Create FSA::Rules object';
ok $r->start, 'Start the machine';
ok $r->curr_state, 'Should have a state';
ok $r->reset, 'Should be able to reset';
```

--:-- base.t Top (1,0) (CPerl Abbrev)----------

# Writing Tests

Emacs

```
use Test::More tests => 5;

use_ok 'FSA::Rules';
ok my $r = FSA::Rules->new,
    'Create FSA::Rules object';
ok $r->start, 'Start the machine';
ok $r->curr_state, 'Should have a state';
ok $r->reset, 'Should be able to reset';
```

--:-- base.t Top (1,0) (CPerl Abbrev)----------

# Writing Tests

Emacs

```
use Test::More tests => 5;

use_ok 'FSA::Rules';
ok my $r = FSA::Rules->new,
    'Create FSA::Rules object';
ok $r->start, 'Start the machine';
ok $r->curr_state, 'Should have a state';
ok $r->reset, 'Should be able to reset';
```

--:-- base.t Top (1,0) (CPerl Abbrev)----------

# Writing Tests

Emacs

```
use Test::More tests => 5;

use_ok 'FSA::Rules';
ok my $r = FSA::Rules->new,
    'Create FSA::Rules object';
ok $r->start, 'Start the machine';
ok $r->curr_state, 'Should have a state';
ok $r->reset, 'Should be able to reset';
```

--:-- base.t Top (1,0) (CPerl Abbrev)----------

# Writing Tests

Emacs

```
use Test::More tests => 5;

use_ok 'FSA::Rules';
ok my $r = FSA::Rules->new,
   'Create FSA::Rules object';
ok $r->start, 'Start the machine';
ok $r->curr_state, 'Should have a state';
ok $r->reset, 'Should be able to reset';
```

**Simple declarative syntax**

--:-- base.t Top (1,0) (CPerl Abbrev)----------

# pgTAP

# pgTAP

- Includes most Test::More functions

# pgTAP

- Includes most Test::More functions
  - ok() — Boolean

# pgTAP

- Includes most Test::More functions
  - ok() — Boolean
  - is() — Value comparison

# pgTAP

- Includes most Test::More functions
  - ok() — Boolean
  - is() — Value comparison
  - isnt() — NOT is()

# pgTAP

- Includes most Test::More functions
  - ok() — Boolean
  - is() — Value comparison
  - isnt() — NOT is()
  - cmp_ok() — Compare with specific operator

# pgTAP

- Includes most Test::More functions
  - ok() — Boolean
  - is() — Value comparison
  - isnt() — NOT is()
  - cmp_ok() — Compare with specific operator
  - matches() — Regex comparison

# pgTAP

- Includes most Test::More functions
  - ok() — Boolean
  - is() — Value comparison
  - isnt() — NOT is()
  - cmp_ok() — Compare with specific operator
  - matches() — Regex comparison
  - imatches() — Case-insensitive regex comparison

# pgTAP

# pgTAP

- Includes Test controls

# pgTAP

- Includes Test controls
  - plan() — How many tests?

# pgTAP

- Includes Test controls
  - plan() — How many tests?
  - no_plan() — Unknown number of tests

# pgTAP

- Includes Test controls
  - plan() — How many tests?
  - no_plan() — Unknown number of tests
  - diag() — Diagnostic output

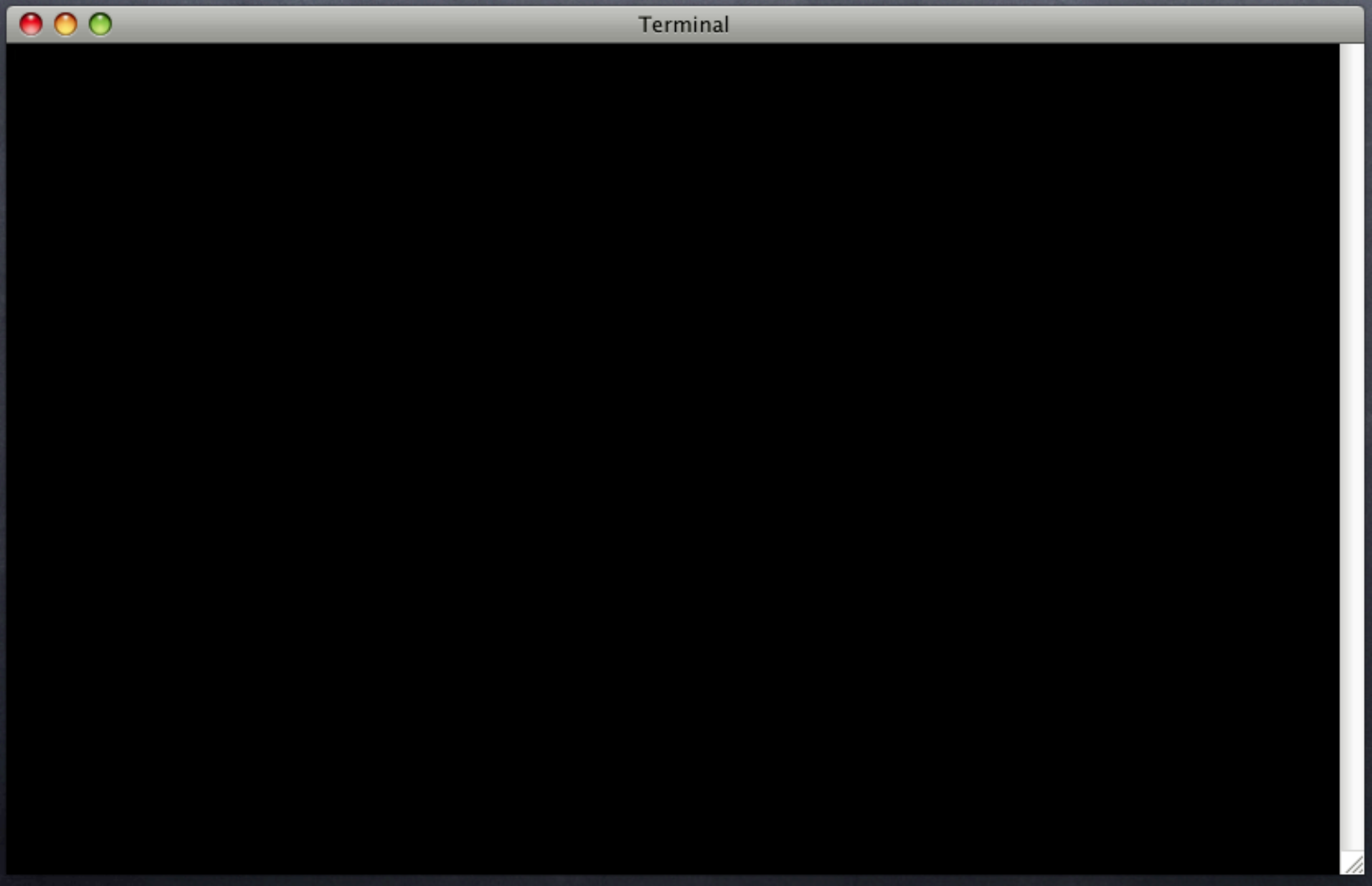
# pgTAP

- Includes Test controls
  - plan() — How many tests?
  - no_plan() — Unknown number of tests
  - diag() — Diagnostic output
  - finish() — Test finished, report!

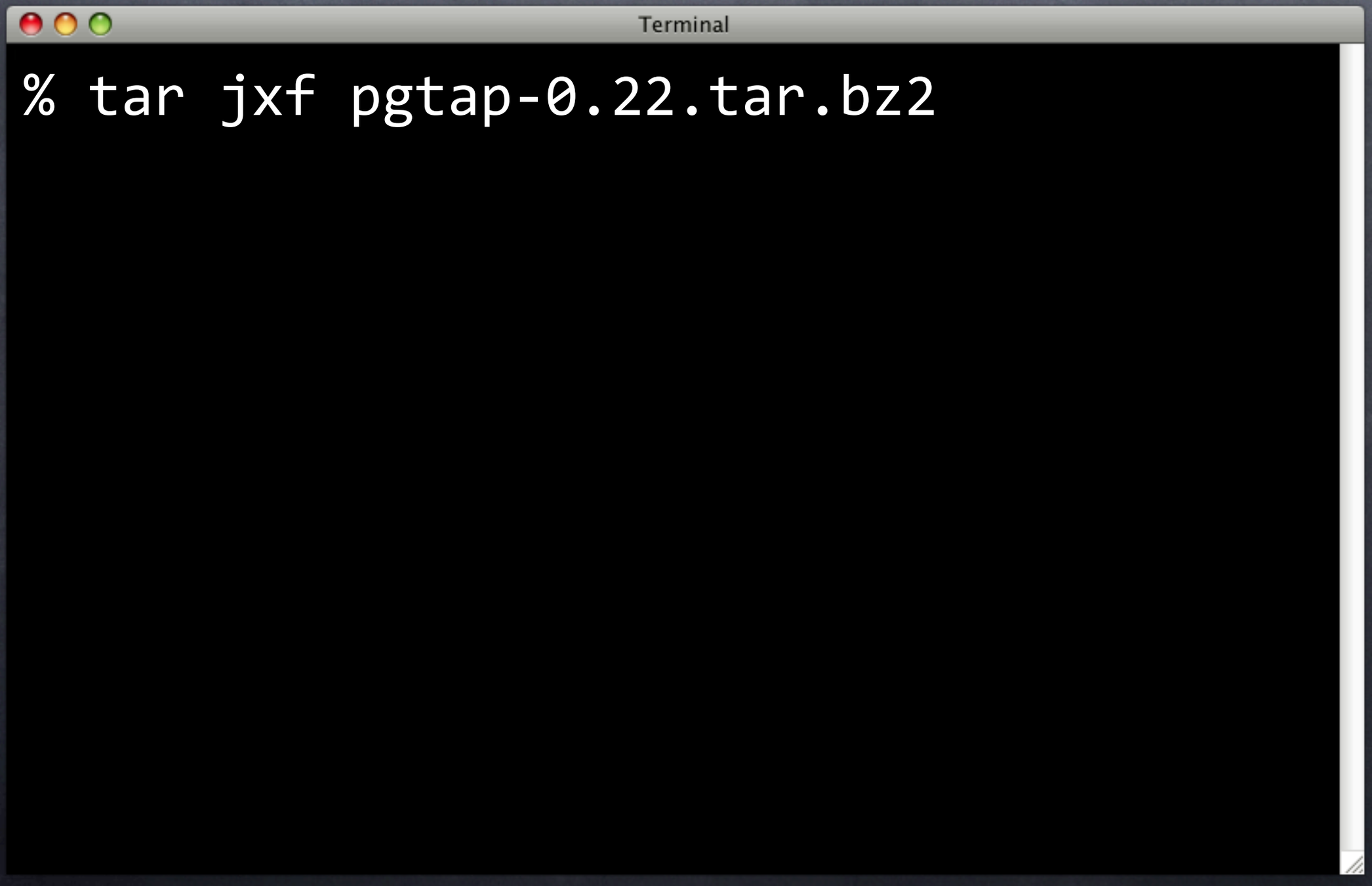
# Follow Along!

http://pgtap.projects.postgresql.org/

# Installing pgTAP



# Installing pgTAP



# Installing pgTAP

# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap USE_PGXS=1
```

# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap USE_PGXS=1
```

## If in contrib...

# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap
```

## If in contrib...

# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap
% sudo make install
```

# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap
% sudo make install
% make installcheck
```

# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap
% sudo make install
% make installcheck
% psql -d template1 -d pgtap.sql
```

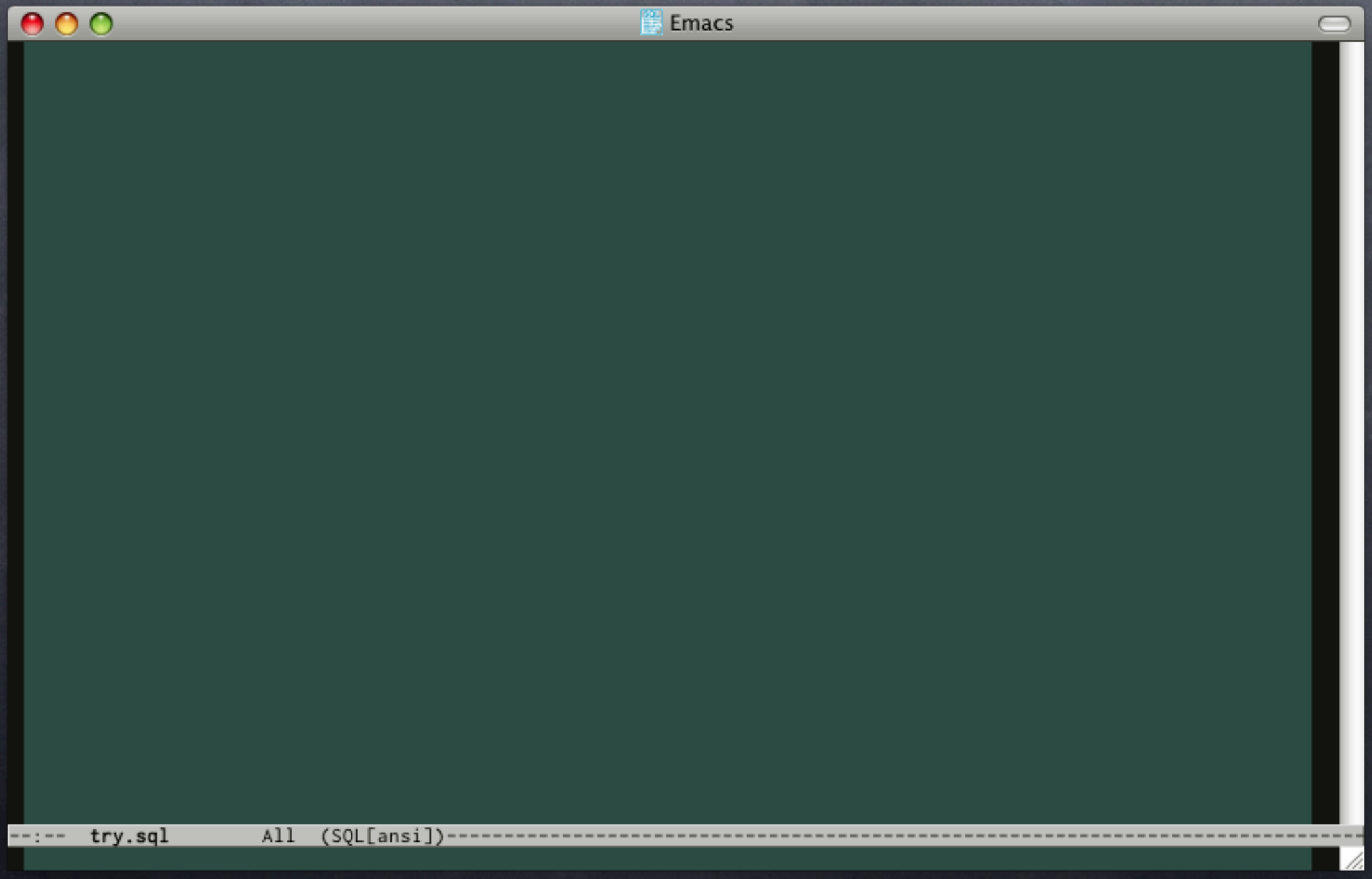
# Installing pgTAP

Terminal

```
% tar jxf pgtap-0.22.tar.bz2
% cd pgtap-0.22
% make TAPSCHEMA=tap
% sudo make install
% make installcheck
% psql -d template1 -d pgtap.sql
```

Need postgresql-dev

# pgTAP Basics



# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# pgTAP Basics

Emacs

```
BEGIN;


-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

If TAPSCHEMA...

```
--:--  try.sql       All  (SQL[ansi])----------
```

# pgTAP Basics

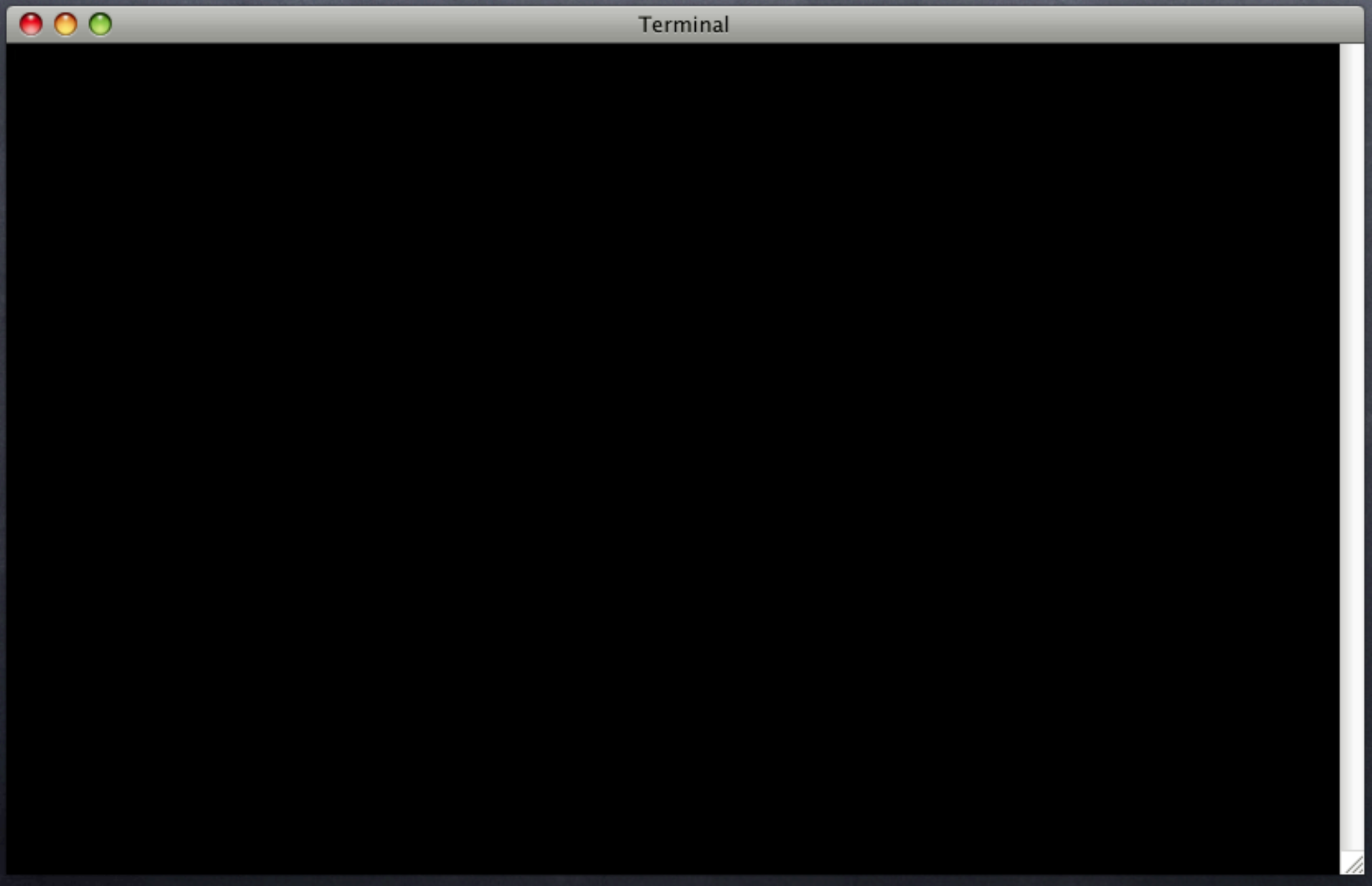
Emacs

```
BEGIN;
SET search_path TO tap, public;

-- Plan the tests.
SELECT plan(1);
--SELECT * FROM no_plan();

SELECT ok( true, 'Truth should be true' );

-- Clean up.
SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

## If TAPSCHEMA...

# Running Tests



# Running Tests

Terminal

```
% pg_prove -v -d try try.sql
try.sql .. psql:try.sql:5: ERROR:  function plan(integer) does
not exist
LINE 1: SELECT plan(1);
               ^
HINT:  No function matches the given name and argument types.
You might need to add explicit type casts.
Dubious, test returned 3 (wstat 768, 0x300)
No subtests run

Test Summary Report
-------------------
try.sql (Wstat: 768 Tests: 0 Failed: 0)
  Non-zero exit status: 3
  Parse errors: No plan found in TAP output
Files=1, Tests=0,  0 wallclock secs ( 0.02 usr +  0.00 sys =
0.02 CPU)
Result: FAIL
```

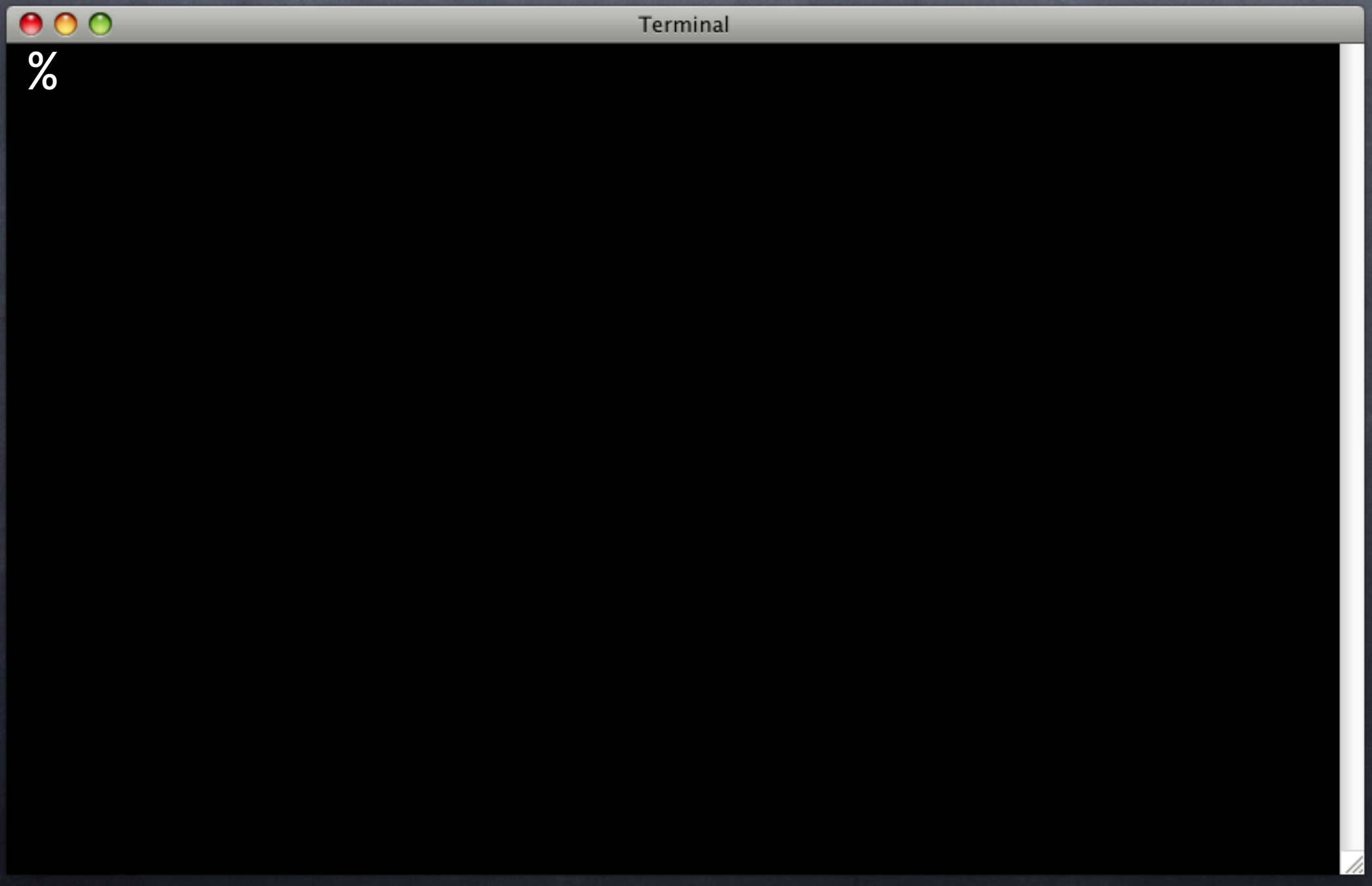
# Running Tests

Terminal

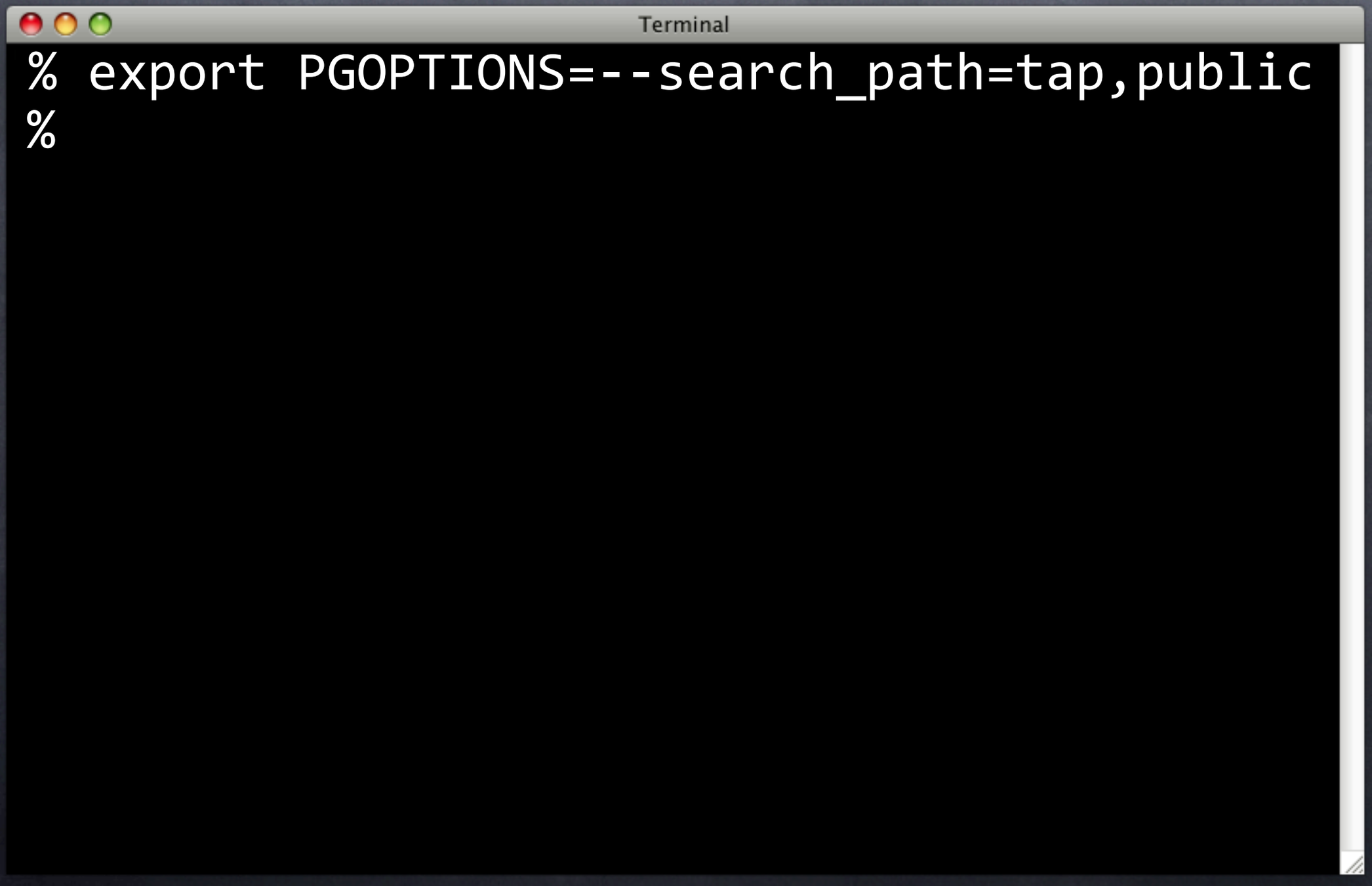
```
% pg_prove -v -d try try.sql
try.sql .. psql:try.sql:5: ERROR:  function plan(integer) does
not exist
LINE 1: SELECT plan(1);
               ^
HINT:  No function matches the given name and argument types.
You might need to add explicit type casts.
Dubious, test returned 3 (wstat 768, 0x300)
No subtests run

Test Summary Report
-------------------
try.sql (Wstat: 768 Tests: 0 Failed: 0)
  Non-zero exit status: 3
  Parse errors: No plan found in TAP output
Files=1, Tests=0,  0 wallclock secs ( 0.02 usr +  0.00 sys =
0.02 CPU)
Result: FAIL
```

Oops

# Running Tests



# Running Tests



# Running Tests

Terminal

```
% export PGOPTIONS=--search_path=tap,public
% pg_prove -v -d try try.sql
try.sql ..
1..1
ok 1 - Truth should be true
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.00 sys = 0.02 CPU)
Result: PASS
```

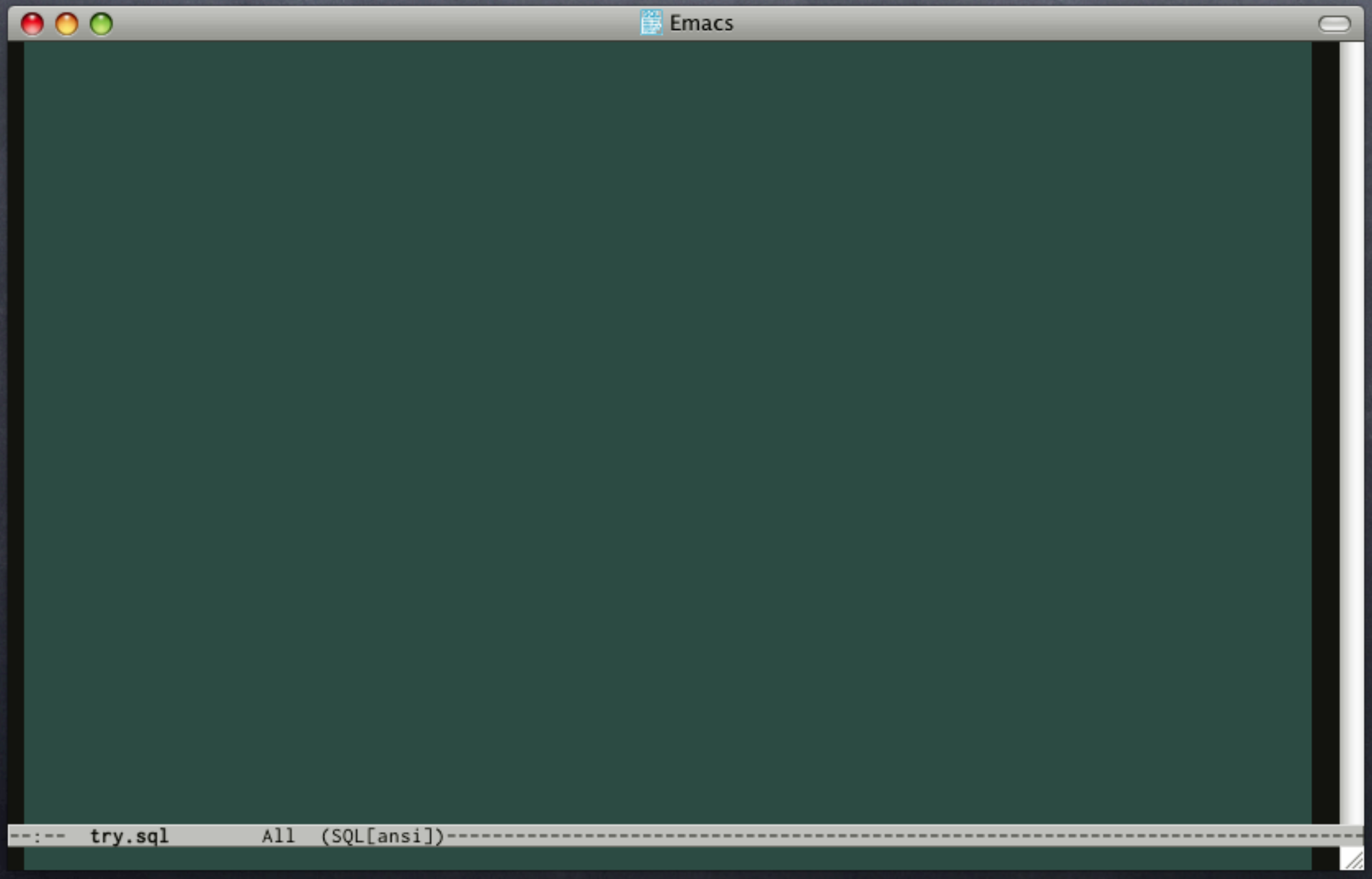
# Running Tests

Terminal

```
% export PGOPTIONS=--search_path=tap,public
% pg_prove -v -d try try.sql
try.sql ..
1..1
ok 1 - Truth should be true
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.00 sys = 0.02 CPU)
Result: PASS
```

WOOt!

# ♥ Failure



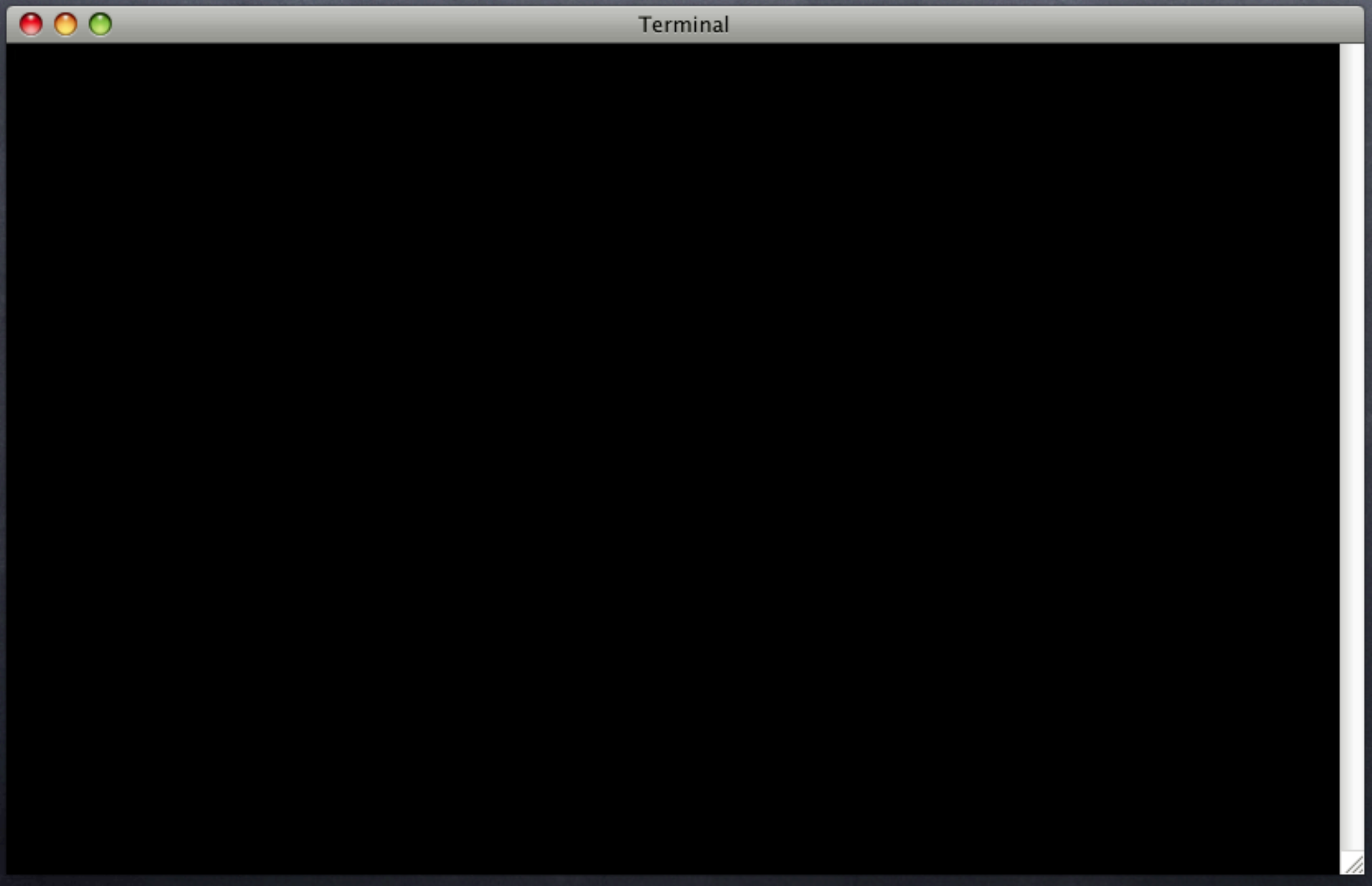
# ♥ Failure

Emacs

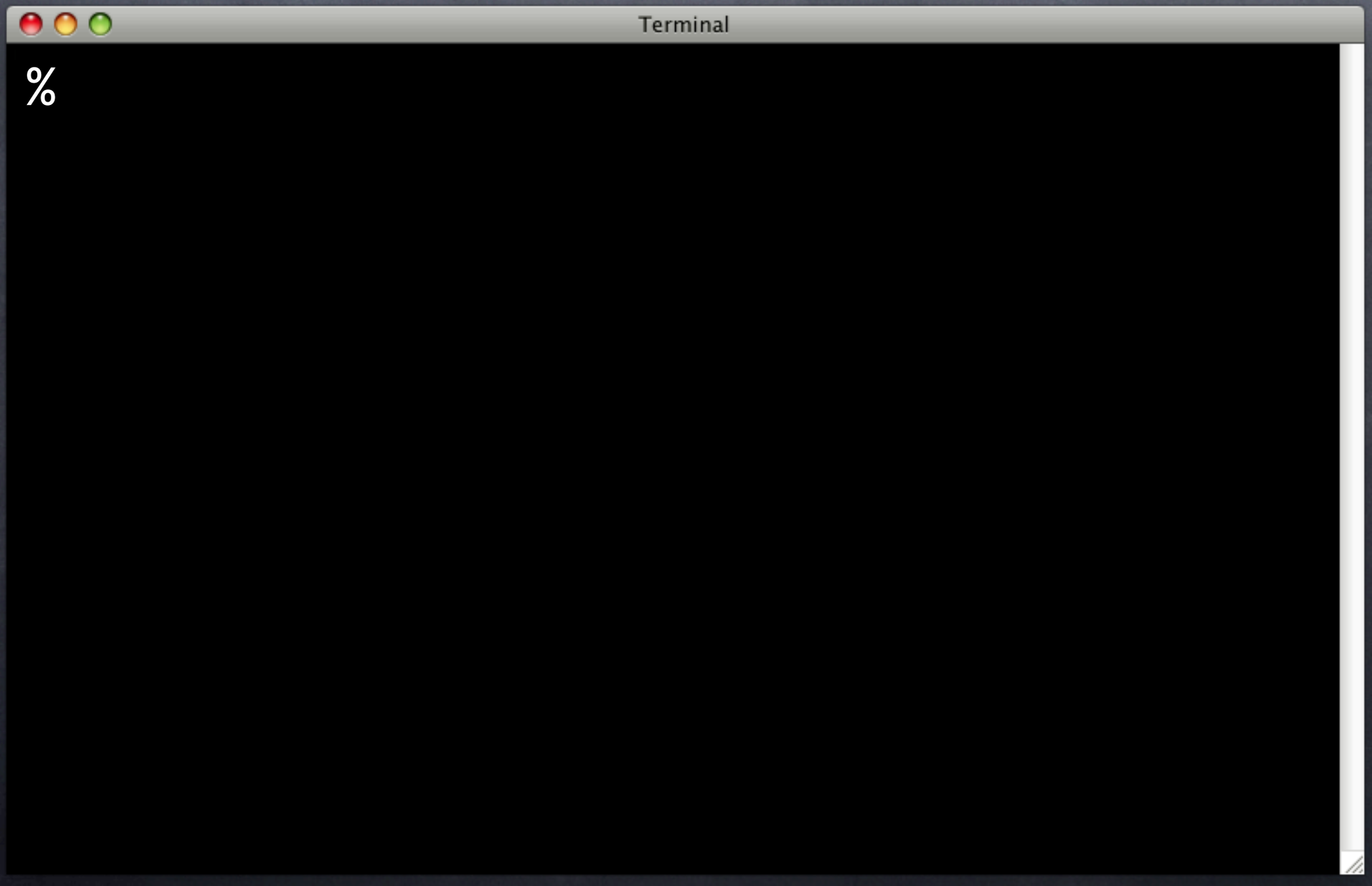
```
BEGIN;

SELECT plan(2);
SELECT ok( 1 + 1 = 2 );
SELECT ok( 2 + 2 = 5 );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# Failing Tests



# Failing Tests



# Failing Tests

Terminal

```
% pg_prove -v -d try fail.sql
fail.sql ..
1..2
ok 1
not ok 2
# Failed test 2
# Looks like you failed 1 test of 2
Failed 1/2 subtests
```

# Failing Tests

Terminal

```
% pg_prove -v -d try fail.sql
fail.sql ..
1..2
ok 1
not ok 2
# Failed test 2
# Looks like you failed 1 test of 2
Failed 1/2 subtests
```

# Failing Tests

Terminal

```
% pg_prove -v -d try fail.sql
fail.sql ..
1..2
ok 1
not ok 2
# Failed test 2
# Looks like you failed 1 test of 2
Failed 1/2 subtests
```

# Failing Tests

Terminal

```
% pg_prove -v -d try fail.sql
fail.sql ..
1..2
ok 1
not ok 2
# Failed test 2
# Looks like you failed 1 test of 2
Failed 1/2 subtests
```

# Failing Tests

Terminal

```
% pg_prove -v -d try fail.sql
fail.sql ..
1..2
ok 1
not ok 2
# Failed test 2
# Looks like you failed 1 test of 2
Failed 1/2 subtests
```

# Scalar Testing

# Scalar Testing

- Where to start testing?

# Scalar Testing

- Where to start testing?
- Start with one unit of code

# Scalar Testing

- Where to start testing?
- Start with one unit of code
- Maybe a custom data type

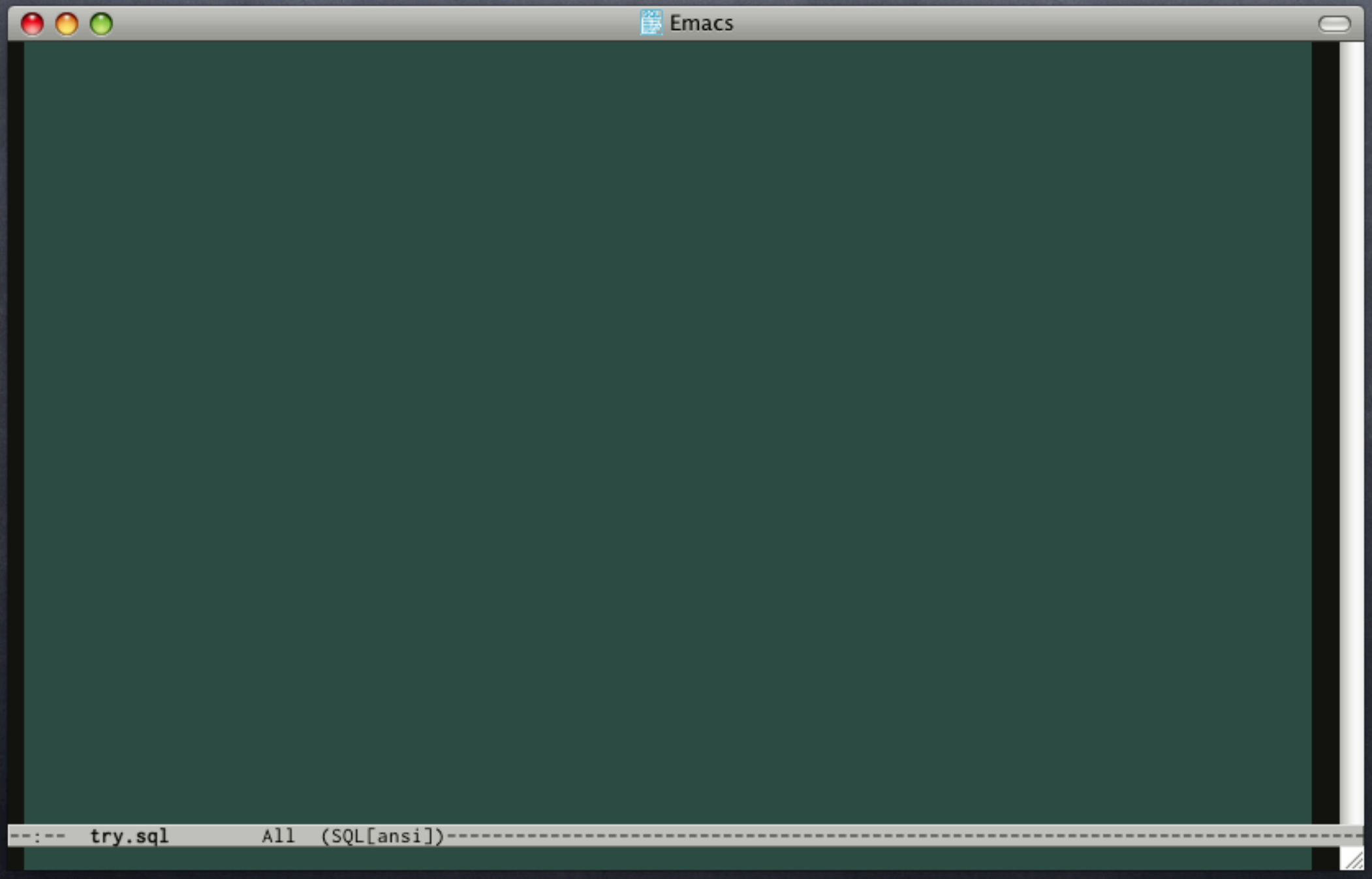
# Scalar Testing

- Where to start testing?
- Start with one unit of code
- Maybe a custom data type
- Obvious candidate for scalar testing

# Scalar Testing

- Where to start testing?
- Start with one unit of code
- Maybe a custom data type
- Obvious candidate for scalar testing
- Testing scalar values

# Testing hstore



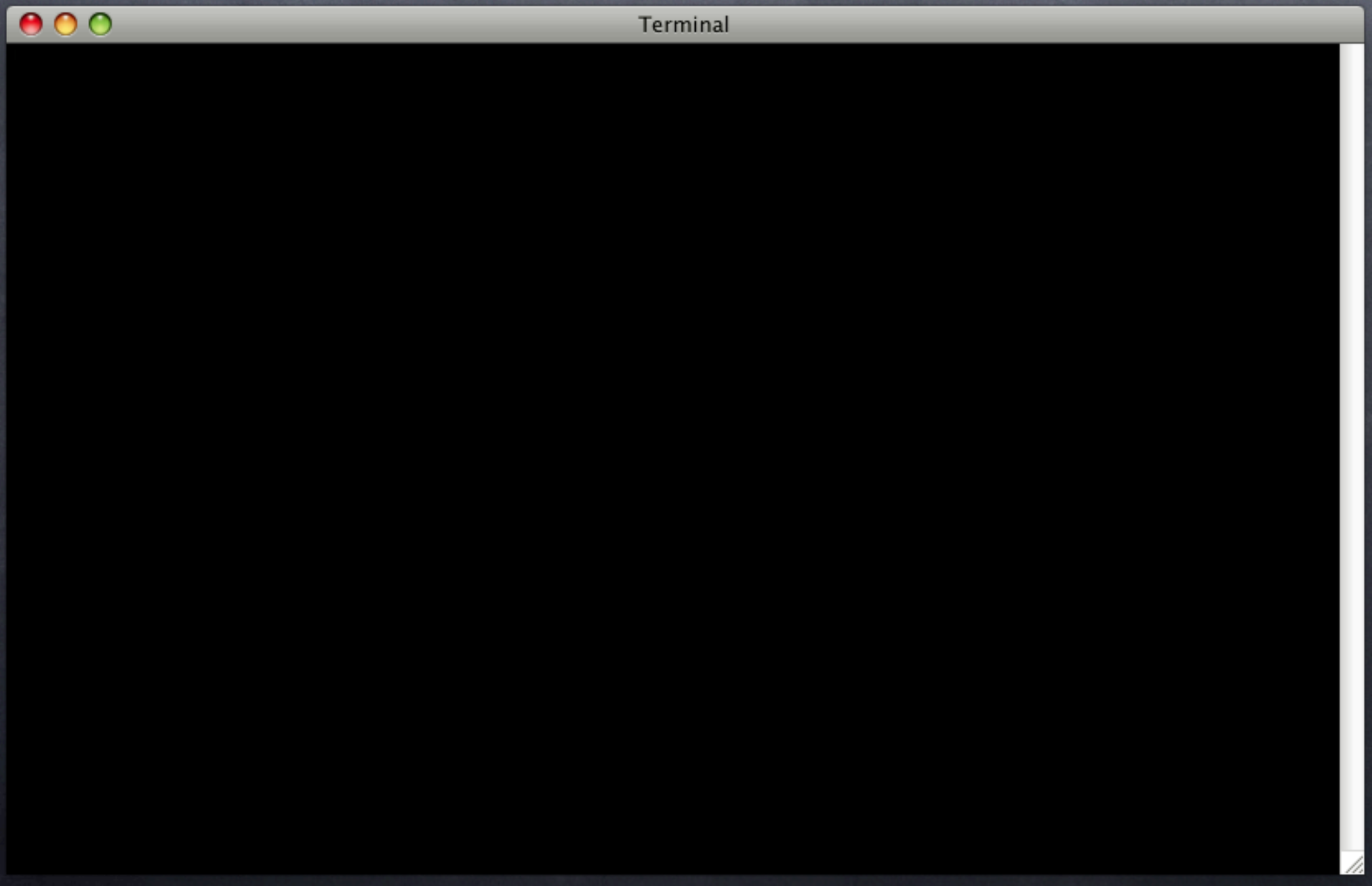
# Testing hstore

Emacs

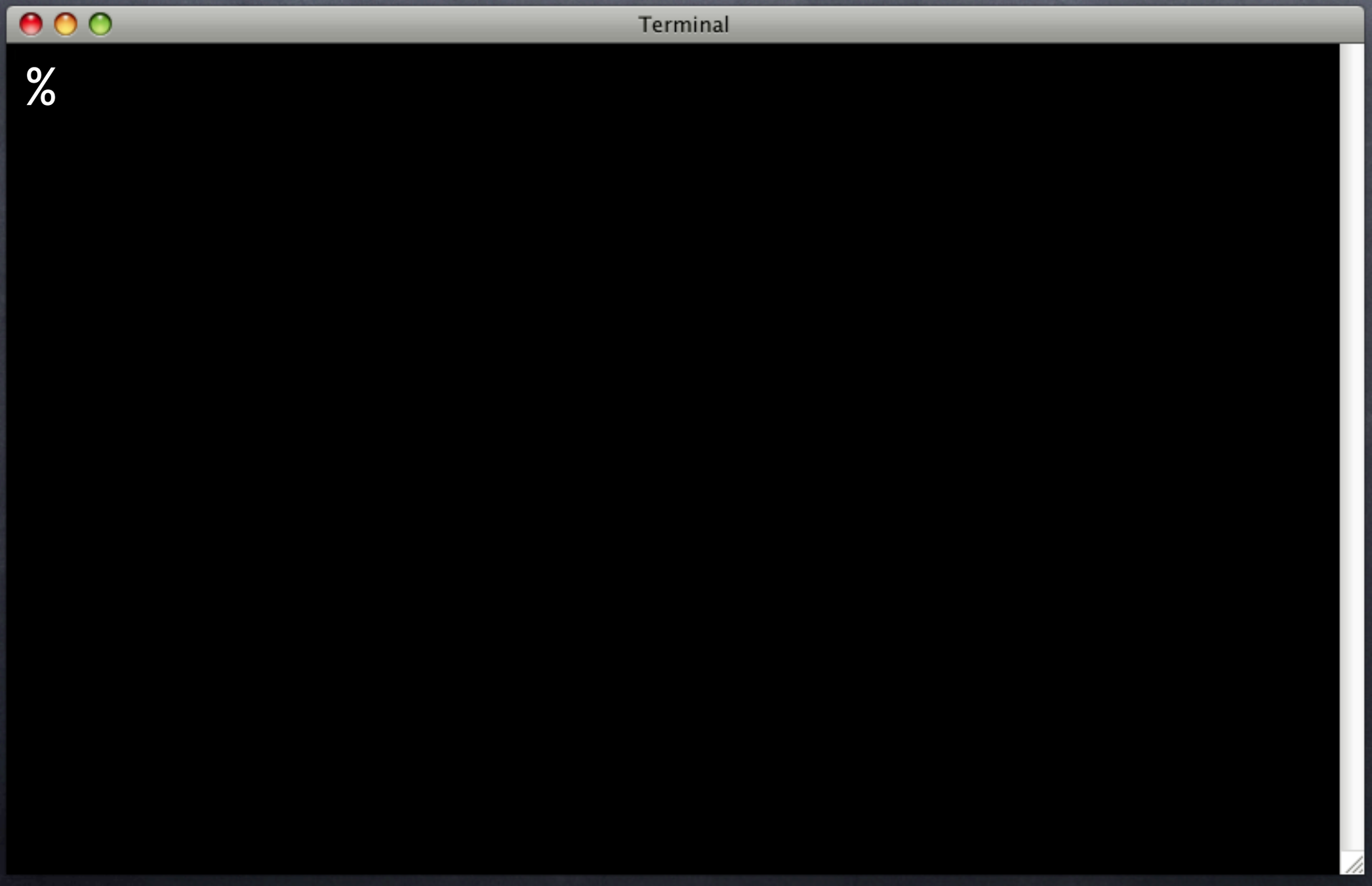
```
BEGIN;

SELECT plan(2);
SELECT ok( 'k => 1'::hstore IS NOT NULL );
SELECT ok(
    pg_typeof('k' => '2') = 'hstore'::regtype
);

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

# Testing hstore



# Testing hstore



# Testing hstore

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..2
ok 1
ok 2
ok
All tests successful.
```

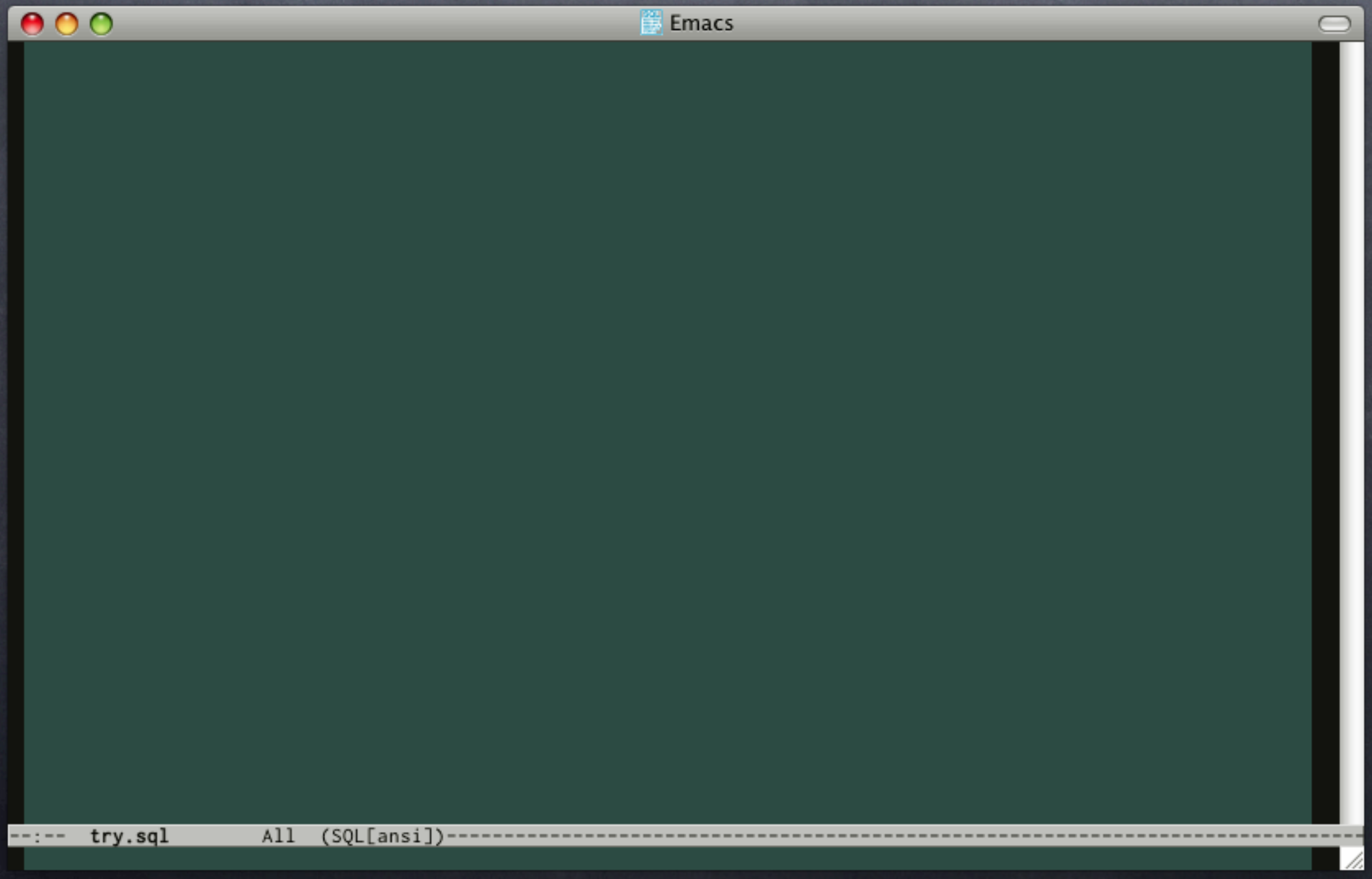
# Testing hstore

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..2
ok 1
ok 2
ok
All tests successful.
```

Your first useful test

# Describe Yourself



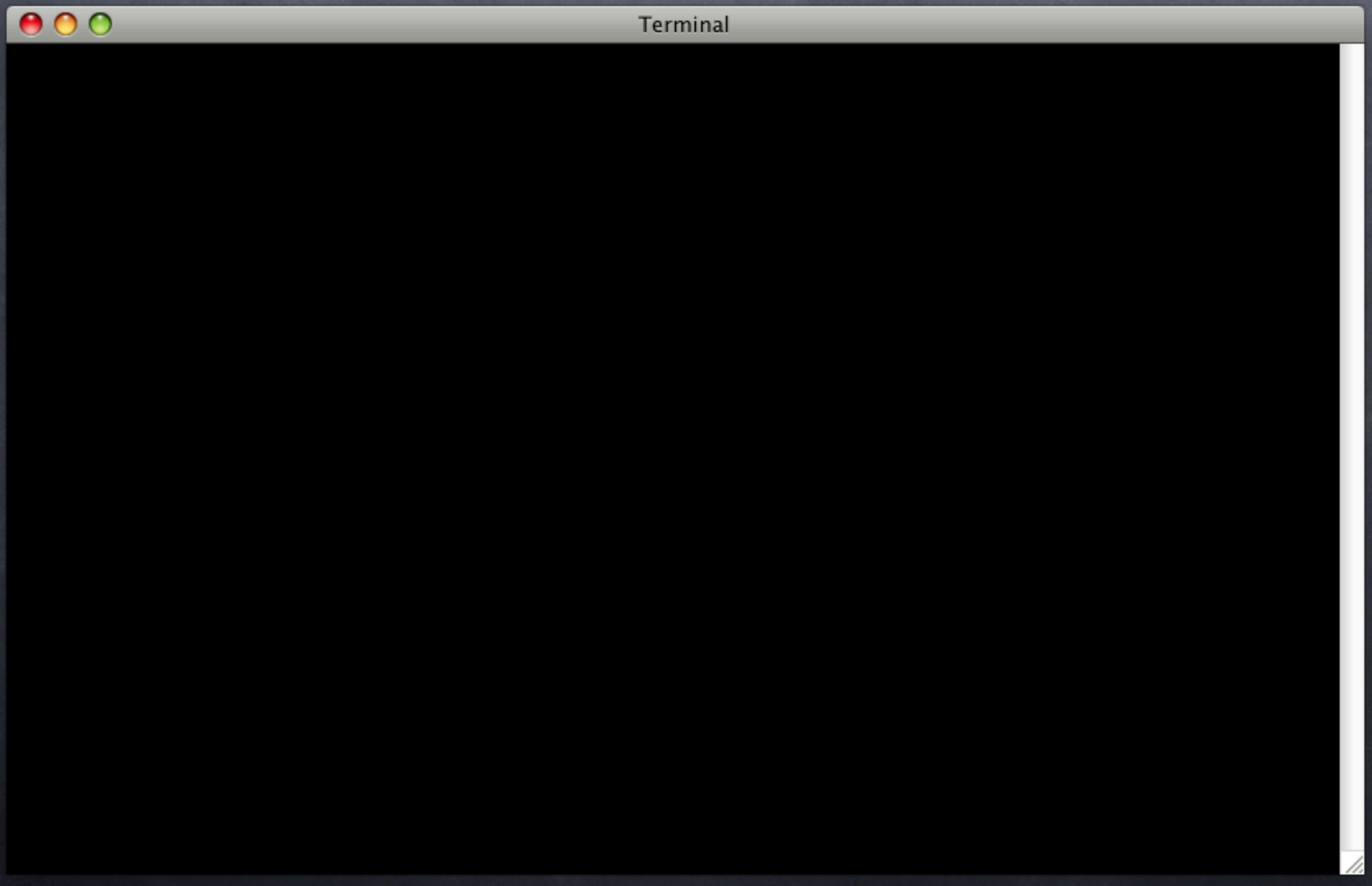
# Describe Yourself

Emacs

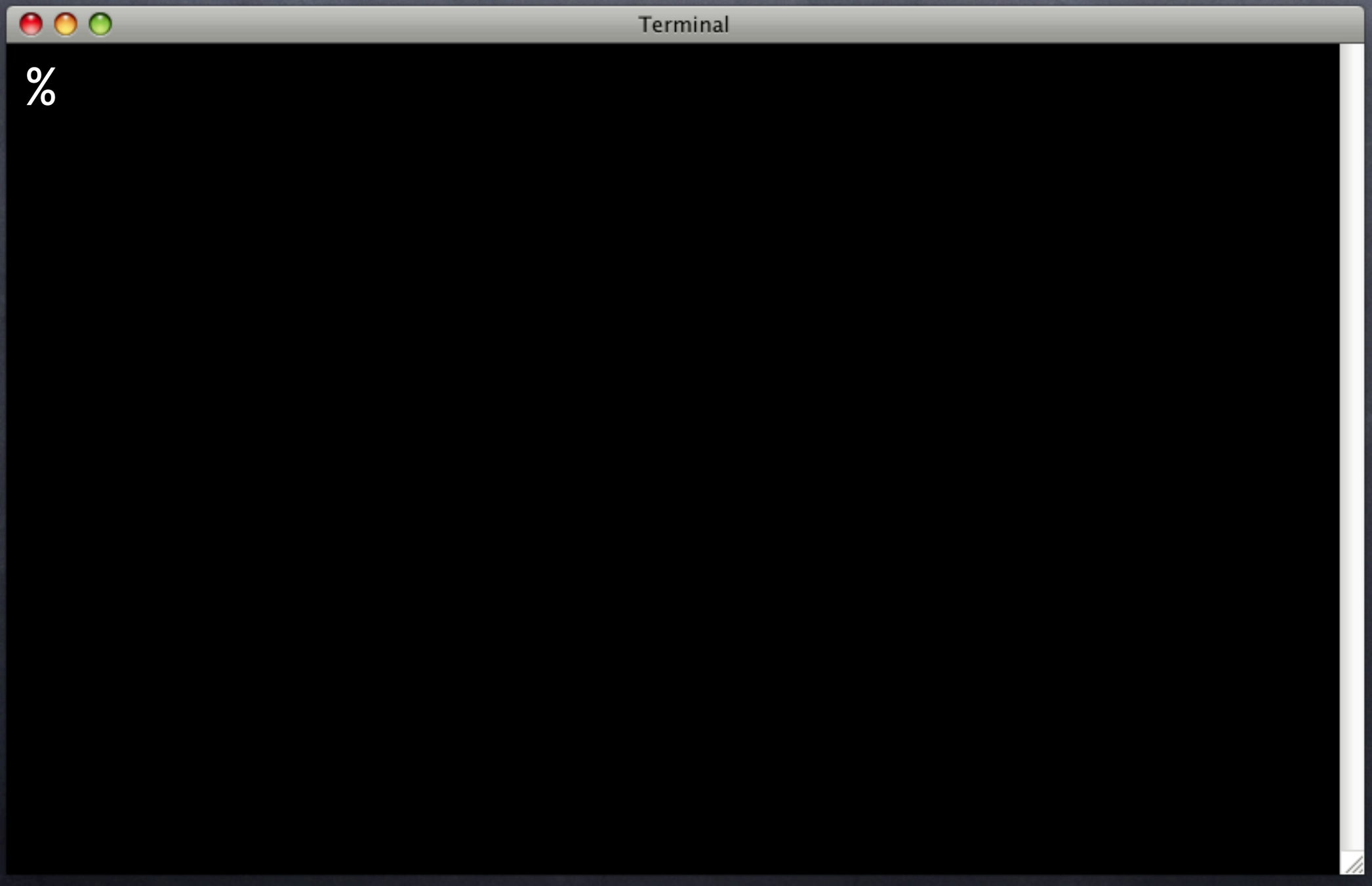
```
SELECT plan(2);
SELECT ok(
    'k => 1'::hstore IS NOT NULL,
    'hstore should work'
);
SELECT ok(
    pg_typeof('k' => '2') = 'hstore'::regtype,
    '=> should create hstore'
);

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Describe Yourself



# Describe Yourself



# Describe Yourself

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..2
ok 1 - hstore should work
ok 2 - => should create hstore
ok
All tests successful.
```

# Simplify, Simplify

Emacs

```
SELECT plan(2);
SELECT ok(
    'k => 1'::hstore IS NOT NULL,
    'hstore should work'
);
SELECT ok(
    pg_typeof('k' => '2') = 'hstore'::regtype,
    '=> should create hstore'
);
SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

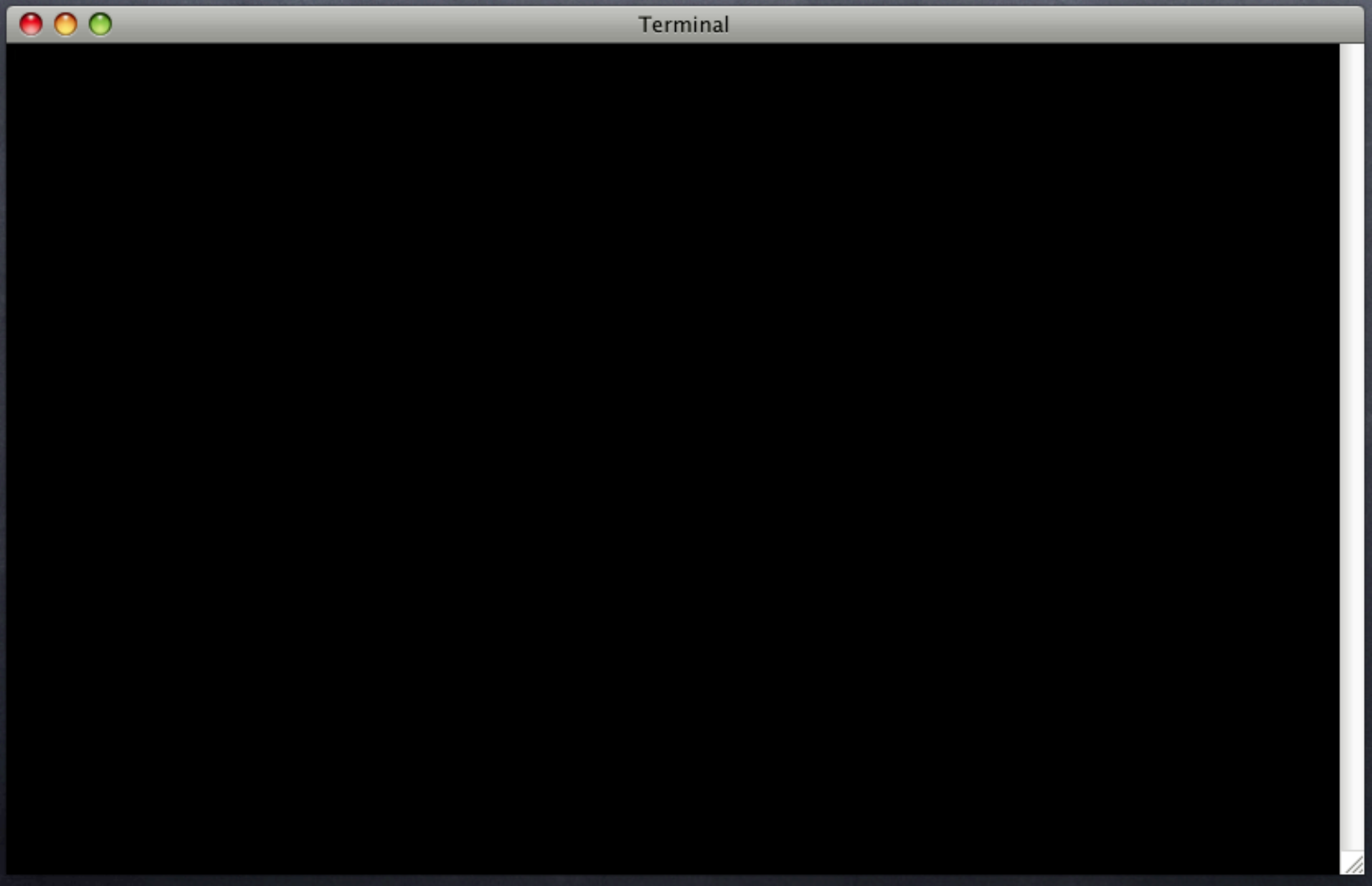
# Simplify, Simplify

Emacs

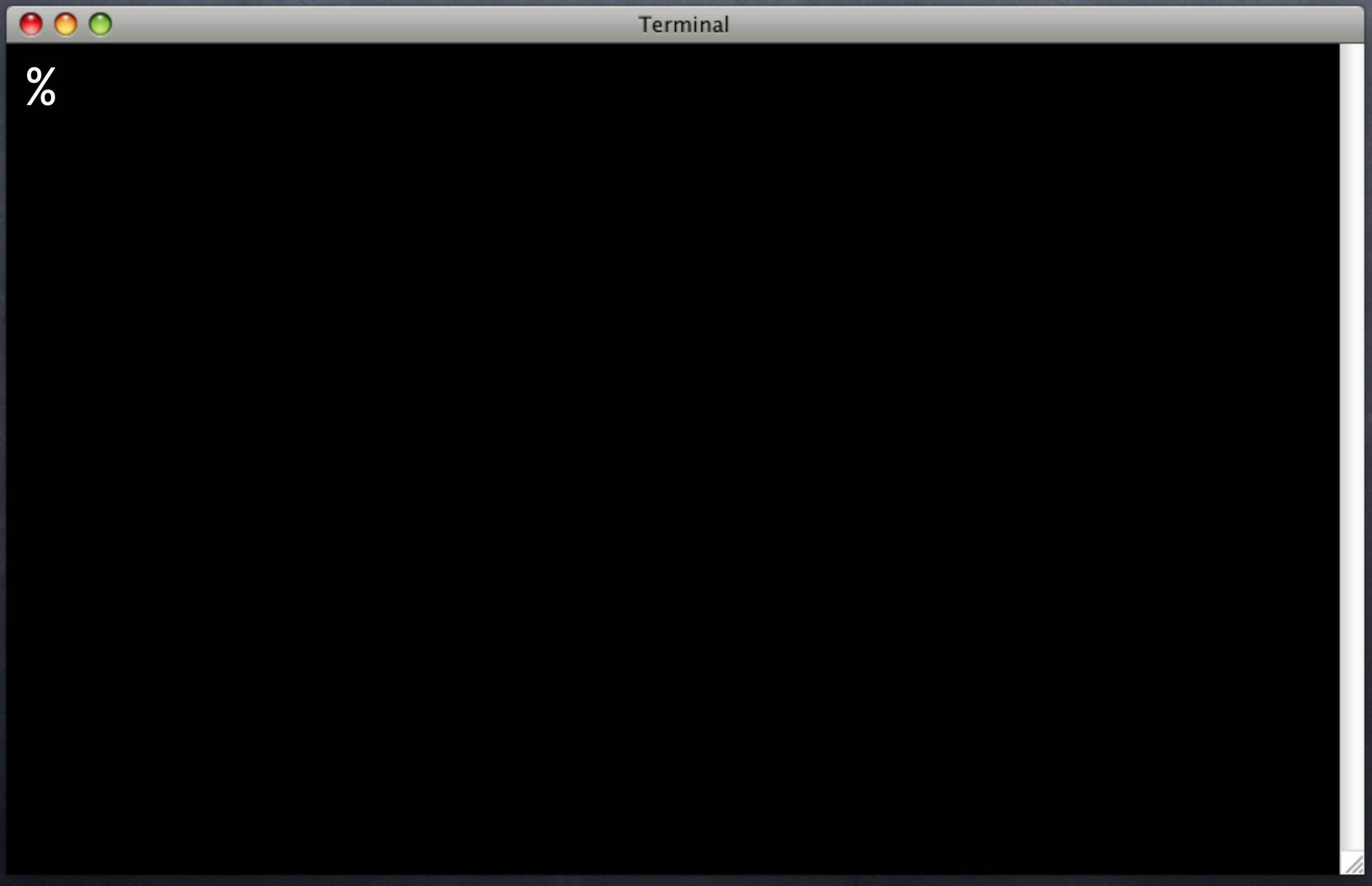
```
SELECT plan(2);
SELECT ok(
    'k => 1'::hstore IS NOT NULL,
    'hstore should work'
);
SELECT isa_ok(
    'k' => '2', 'hstore', '=> return value'
);
SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Simplify, Simplify



# Simplify, Simplify

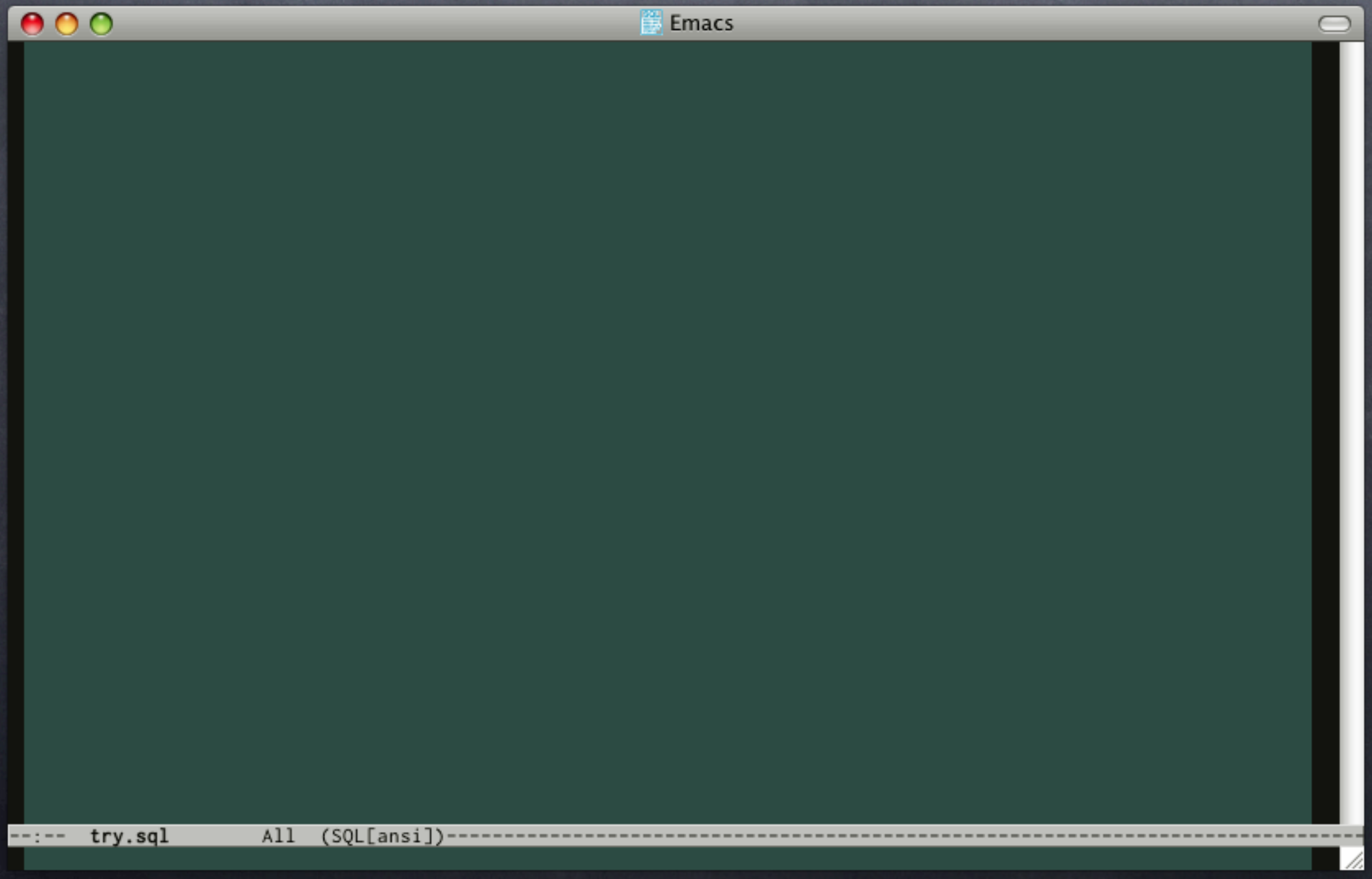


# Simplify, Simplify

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..2
ok 1 - hstore should work
ok 2 - => return value isa hstore
ok
All tests successful.
```

# Test the Manual



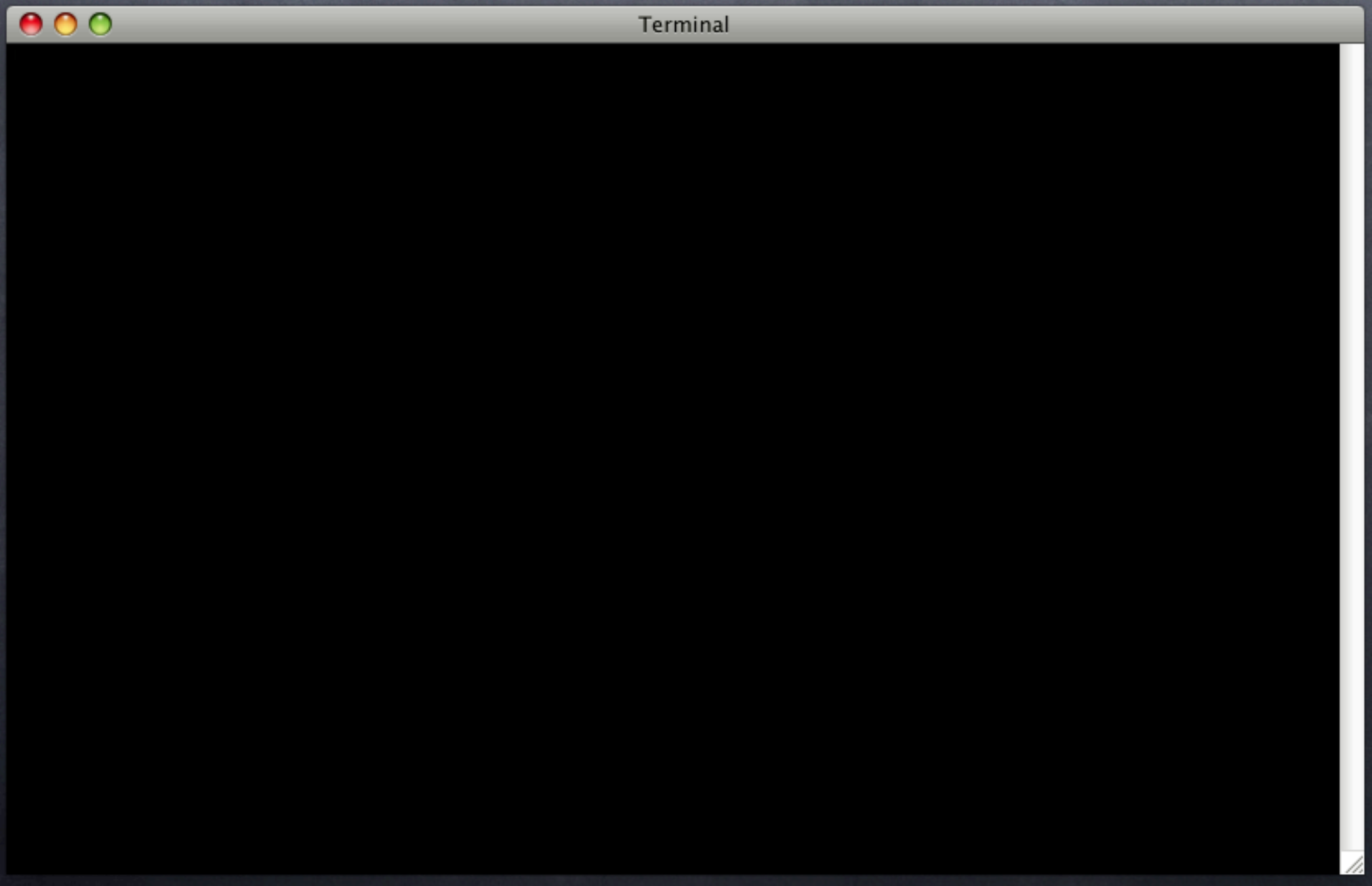
# Test the Manual

Emacs

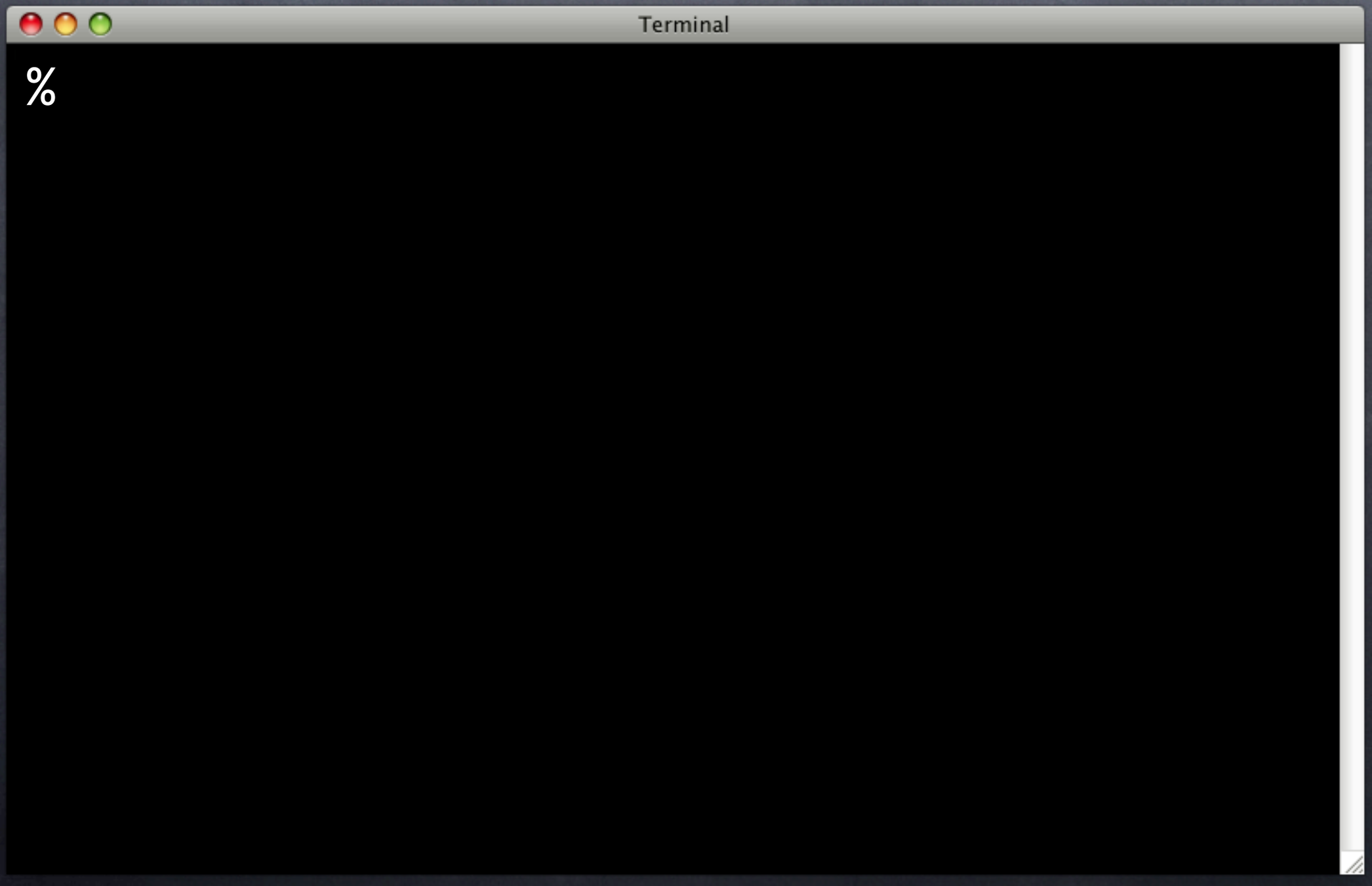
```
SELECT plan(6);

SELECT ok( 'a=>x, b=>y'::hstore -> 'a' = 'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT ok( akeys('a=>1,b=>2') = '{a,c}', 'akeys()' );
SELECT ok( avals('a=>1,b=>2') = '{1,2}', 'avals()' );

SELECT * FROM finish();
```

```
--:--  try.sql       All  (SQL[ansi])----------
```

# Test the Manual



# Test the Manual



# Test the Manual

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
not ok 5 - akeys()
# Failed test 5: "akeys()"
ok 6 - avals()
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Test the Manual

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
not ok 5 - akeys()
# Failed test 5: "akeys()"
ok 6 - avals()
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

Um, why?

# What is() It?

Emacs

```
SELECT plan(6);

SELECT ok( 'a=>x, b=>y'::hstore -> 'a' = 'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT ok( akeys('a=>1,b=>2') = '{a,c}', 'akeys()' );
SELECT ok( avals('a=>1,b=>2') = '{1,2}', 'avals()' );

SELECT * FROM finish();
```

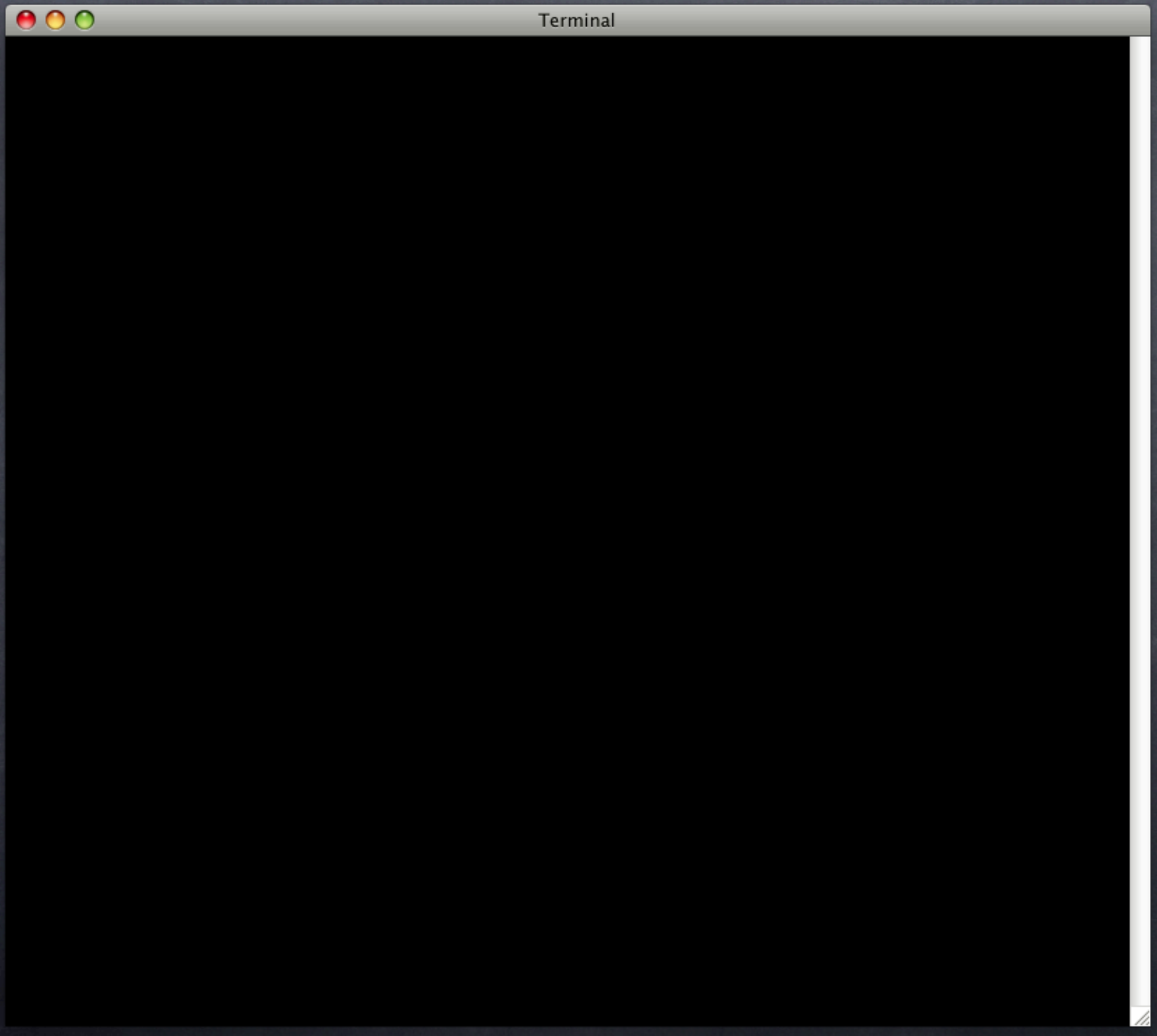
--:--  try.sql  All  (SQL[ansi])----------

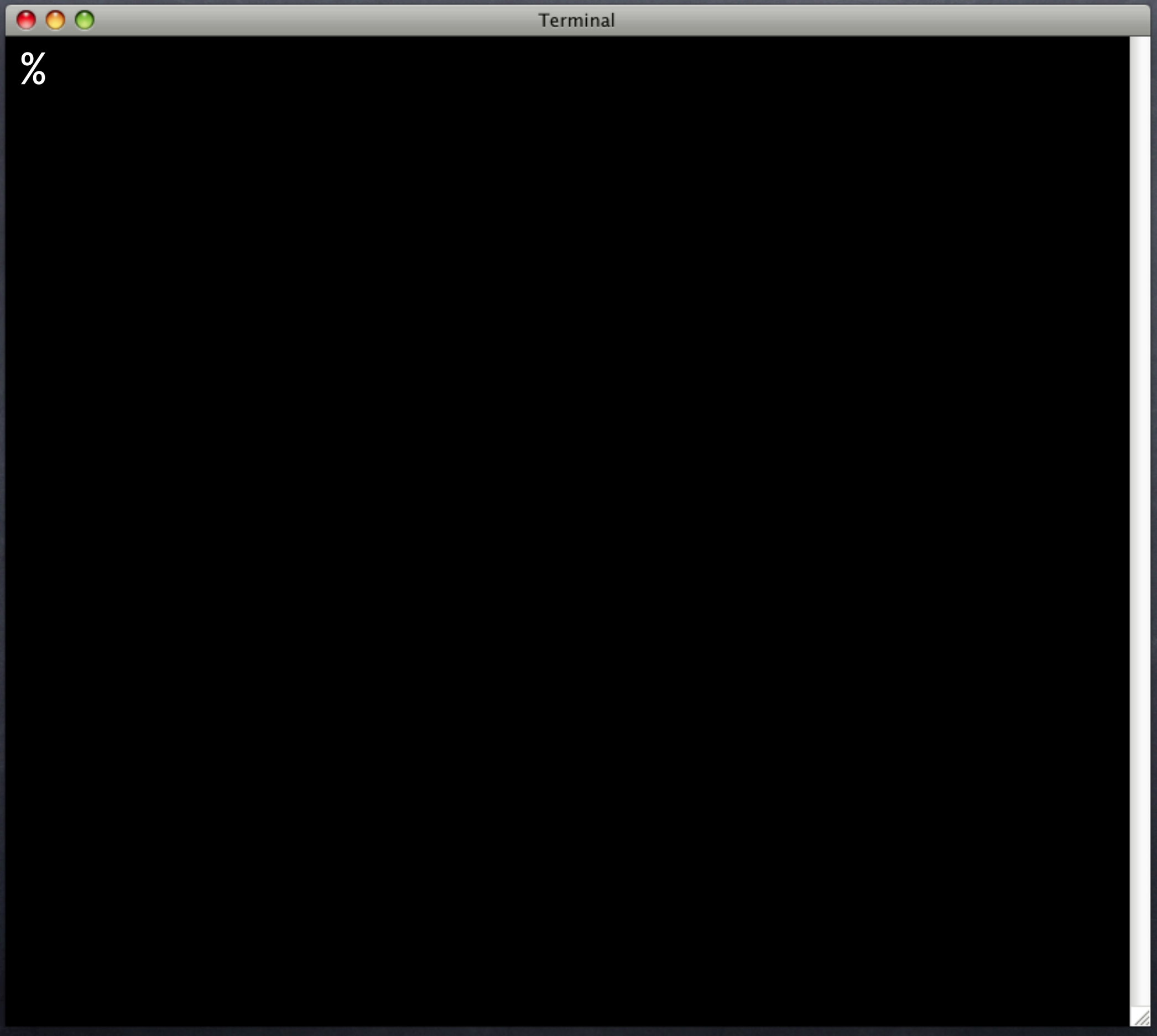
# What is() It?

Emacs

```
SELECT plan(6);

SELECT is( 'a=>x, b=>y'::hstore -> 'a',  'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT is( akeys('a=>1,b=>2'),  '{a,c}', 'akeys()' );
SELECT is( avals('a=>1,b=>2'),  '{1,2}', 'avals()' );

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```


Terminal


Terminal
%

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
not ok 5 - akeys()
# Failed test 5: "akeys()"
#          have: {a,b}
#          want: {a,c}
ok 6 - avals()
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
not ok 5 - akeys()
# Failed test 5: "akeys()"
#          have: {a,b}
#          want: {a,c}
ok 6 - avals()
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
not ok 5 - akeys()
# Failed test 5: "akeys()"
#          have: {a,b}
#          want: {a,c}
ok 6 - avals()
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# There it is

# What is() It?

Emacs

```
SELECT plan(6);

SELECT is( 'a=>x, b=>y'::hstore -> 'a',  'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT is( akeys('a=>1,b=>2'), '{a,c}', 'akeys()' );
SELECT is( avals('a=>1,b=>2'), '{1,2}', 'avals()' );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# What is() It?

Emacs

```
SELECT plan(6);

SELECT is( 'a=>x, b=>y'::hstore -> 'a',  'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT is( akeys('a=>1,b=>2'), '{a,c}', 'akeys()' );
SELECT is( avals('a=>1,b=>2'), '{1,2}', 'avals()' );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# What is() It?

Emacs

```
SELECT plan(6);

SELECT is( 'a=>x, b=>y'::hstore -> 'a',  'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT is( akeys('a=>1,b=>2'), '{a,c}', 'akeys()' );
SELECT is( avals('a=>1,b=>2'), '{1,2}', 'avals()' );

SELECT * FROM finish();
```

```
--:--  try.sql       All  (SQL[ansi])----------
```

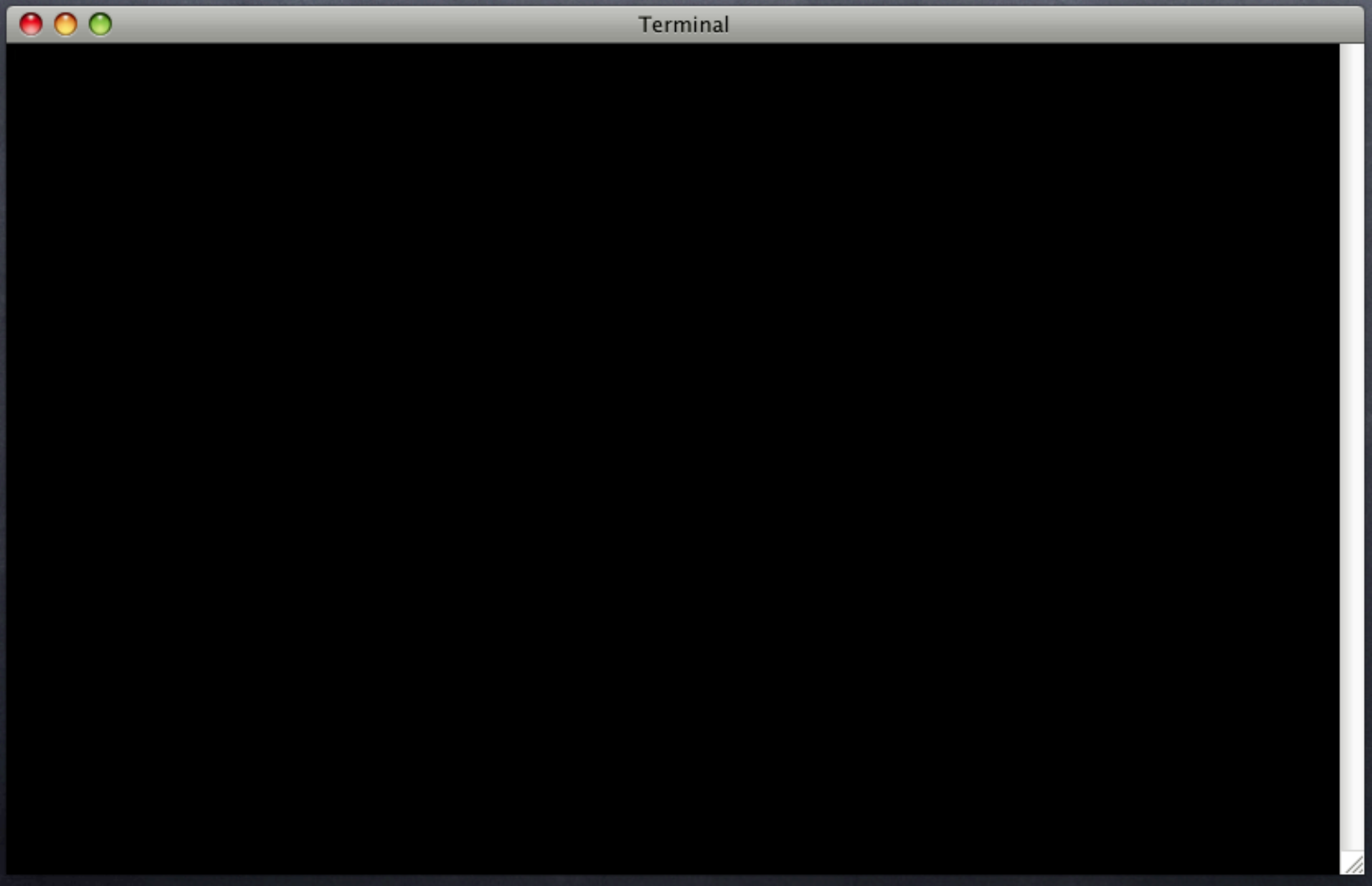
# What is() It?

Emacs

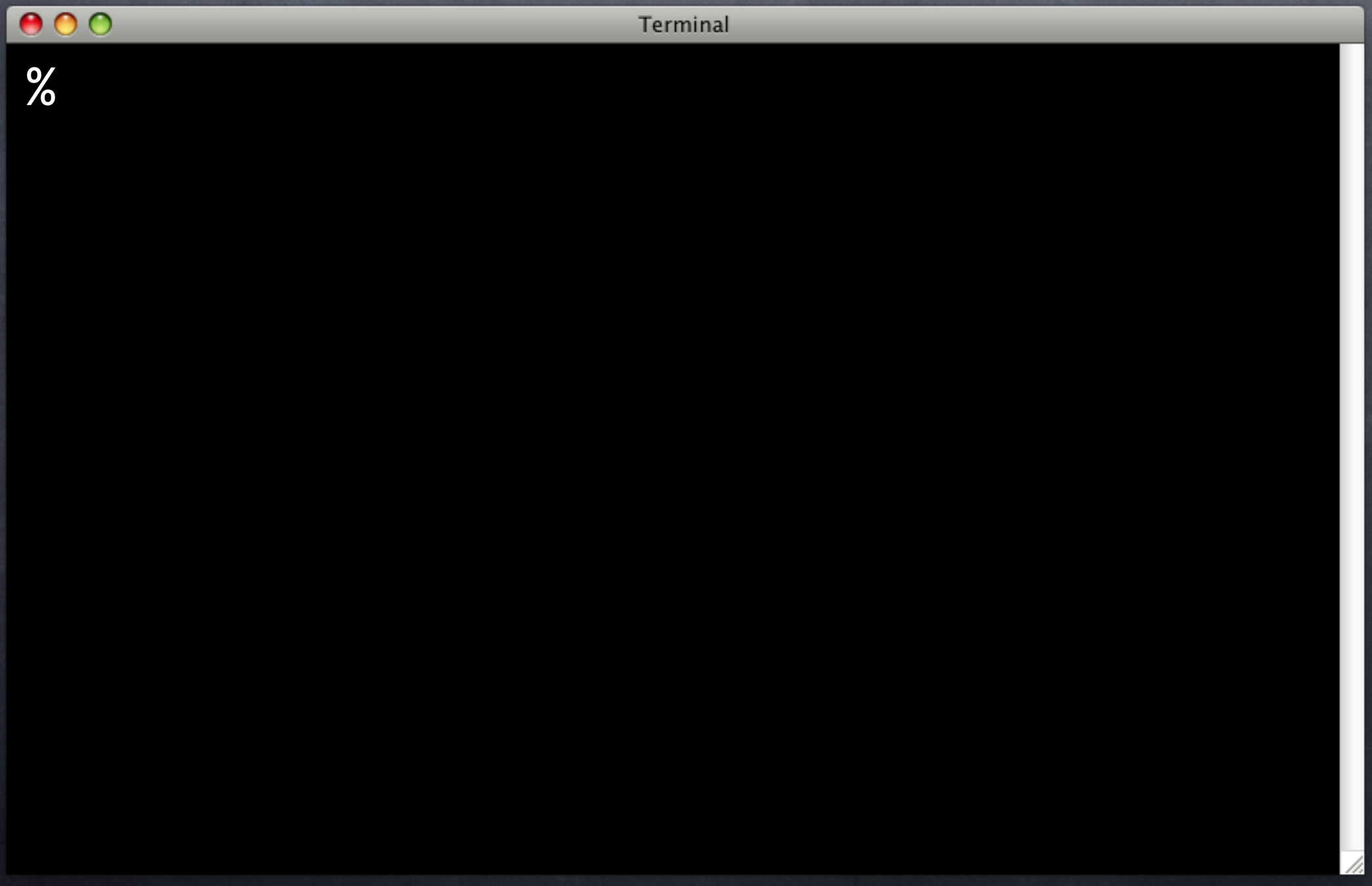
```
SELECT plan(6);

SELECT is( 'a=>x, b=>y'::hstore -> 'a',  'x', 'op ->' );
SELECT ok( 'a=>1'::hstore ? 'a', 'op ?' );
SELECT ok( 'a=>b, b=>1'::hstore @> 'b=>1', 'op @>' );
SELECT ok( NOT 'a=>c'::hstore <@ 'a=>b, b=>1', 'op <@' );
SELECT is( akeys('a=>1,b=>2'), '{a,b}', 'akeys()' );
SELECT is( avals('a=>1,b=>2'), '{1,2}', 'avals()' );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# What is() It?



# What is() It?



# What is() It?

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
ok 5 - akeys()
ok 6 - avals()
ok
All tests successful.
```

# What is() It?

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..6
ok 1 - op ->
ok 2 - op ?
ok 3 - op @>
ok 4 - op <@
ok 5 - akeys()
ok 6 - avals()
ok
All tests successful.
```

Much better!

# Repetition

# Repetition

- Want to test lots of values

# Repetition

- Want to test lots of values
- Try to trick code with edge cases

# Repetition

- Want to test lots of values
- Try to trick code with edge cases
- Writing same tests with different values

# Repetition

- Want to test lots of values
- Try to trick code with edge cases
- Writing same tests with different values
- Could get old fast

# Repetition

- Want to test lots of values
- Try to trick code with edge cases
- Writing same tests with different values
- Could get old fast
- Use a table for repetitive tests

Emacs

--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

--:--  try.sql       All  (SQL[ansi])----------

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,    -- val -> arrow_rop
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,    -- val -> arrow_rop
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,    -- val -> arrow_rop
    arrow_val text,    --      = arrow_val
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,    -- val ? qmark_rop
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[], -- akeys(val) = akeys_val
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]  -- avals(val) = avals_val
);
RESET client_min_messages;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SET client_min_messages = warning;
CREATE TEMPORARY TABLE hs_tests (
    val hstore,
    arrow_rop text,
    arrow_val text,
    qmark_rop text,
    akeys_val text[],
    avals_val text[]
);
RESET client_min_messages;

-- Insert test values.
INSERT INTO hs_tests VALUES
('a=>1,b=>2', 'a', '1', 'a', '{a,b}', '{1,2}'),
('a=>1',      'a', '1', 'a', '{a}',   '{1}')
;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql       All  (SQL[ansi])----------
```

Emacs

```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

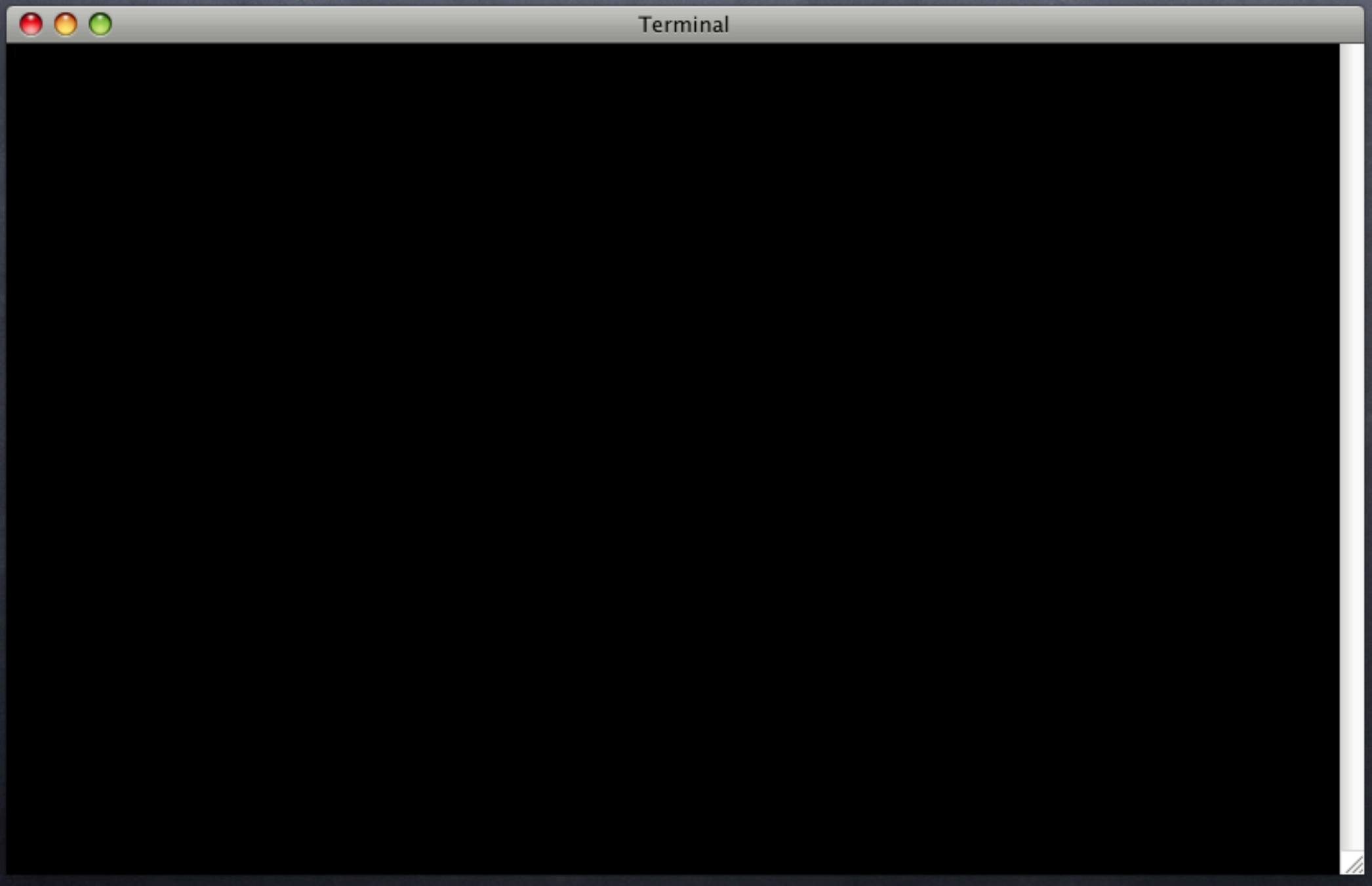
--:--  try.sql  All  (SQL[ansi])----------

Emacs

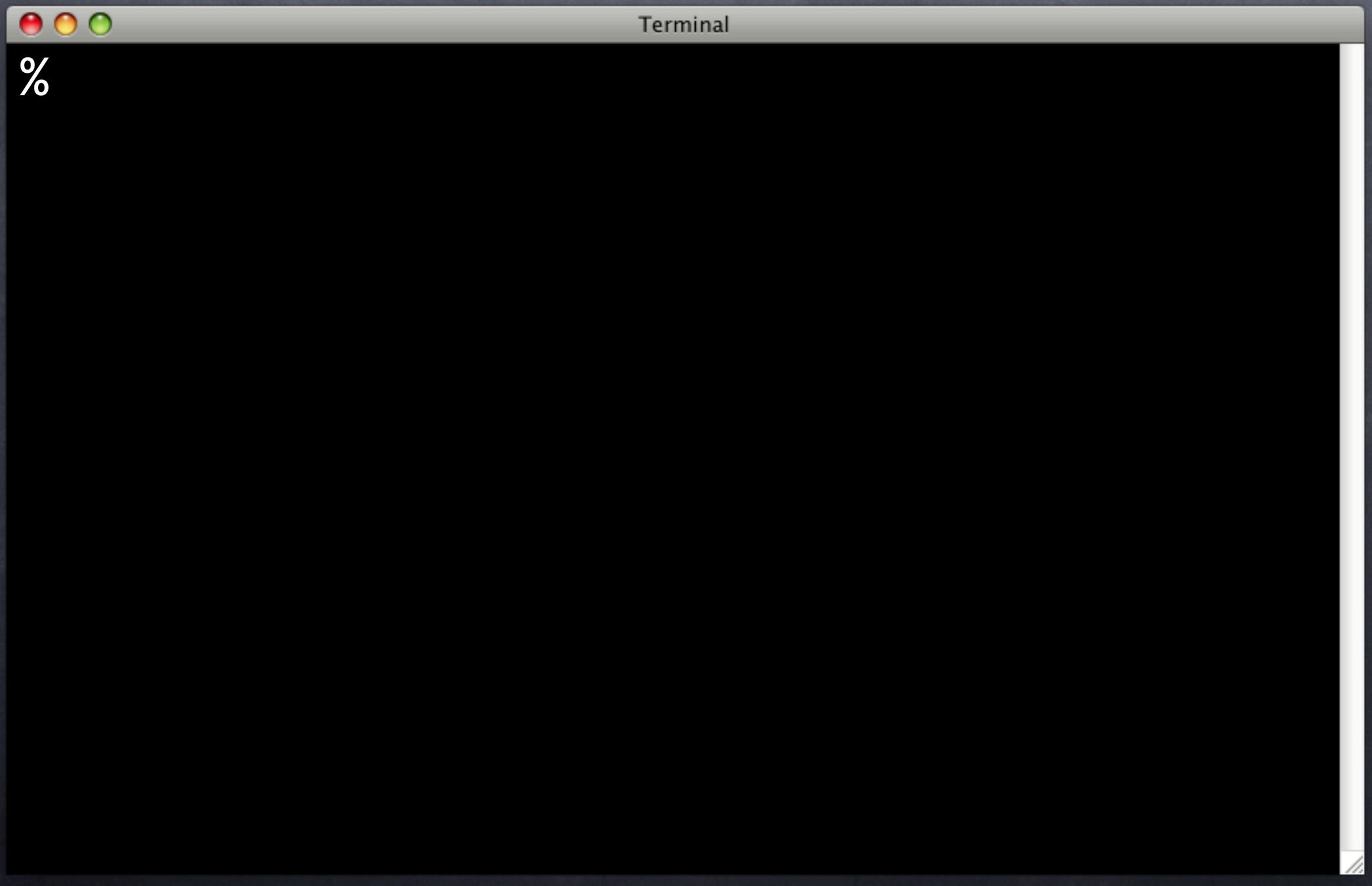
```
SELECT plan(COUNT(*)::int * 4) FROM hs_tests;

SELECT is(
    val -> arrow_rop,
    arrow_val,
    quote_literal(val) || ' ? ' || quote_literal(arrow_val)
) FROM hs_tests;

SELECT ok(
    val ? qmark_rop,
    quote_literal(val) || ' -> ' || quote_literal(qmark_rop)
) FROM hs_tests;

SELECT is(
    akeys(val), akeys_val,
    'akeys(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT is(
    avals(val), avals_val,
    'avals(' || quote_literal(val) || ')'
) FROM hs_tests;

SELECT * FROM finish();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Repetition



# Repetition



# Repetition

Terminal

```
% pg_prove -v -d try hstore.sql
hstore.sql ..
1..8
ok 1 - '"a"=>"1", "b"=>"2"' ? '1'
ok 2 - '"a"=>"1"' ? '1'
ok 3 - '"a"=>"1", "b"=>"2"' -> 'a'
ok 4 - '"a"=>"1"' -> 'a'
ok 5 - akeys('"a"=>"1", "b"=>"2"')
ok 6 - akeys('"a"=>"1"')
ok 7 - avals('"a"=>"1", "b"=>"2"')
ok 8 - avals('"a"=>"1"')
ok
All tests successful.
```

# Add Tests

Emacs

```
-- Insert test values.
INSERT INTO hs_tests VALUES
('a=>1,b=>2',  'a',   '1',   'a',   '{a,b}', '{1,2}'),
('a=>1',       'a',   '1',   'a',   '{a}',   '{1}')
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

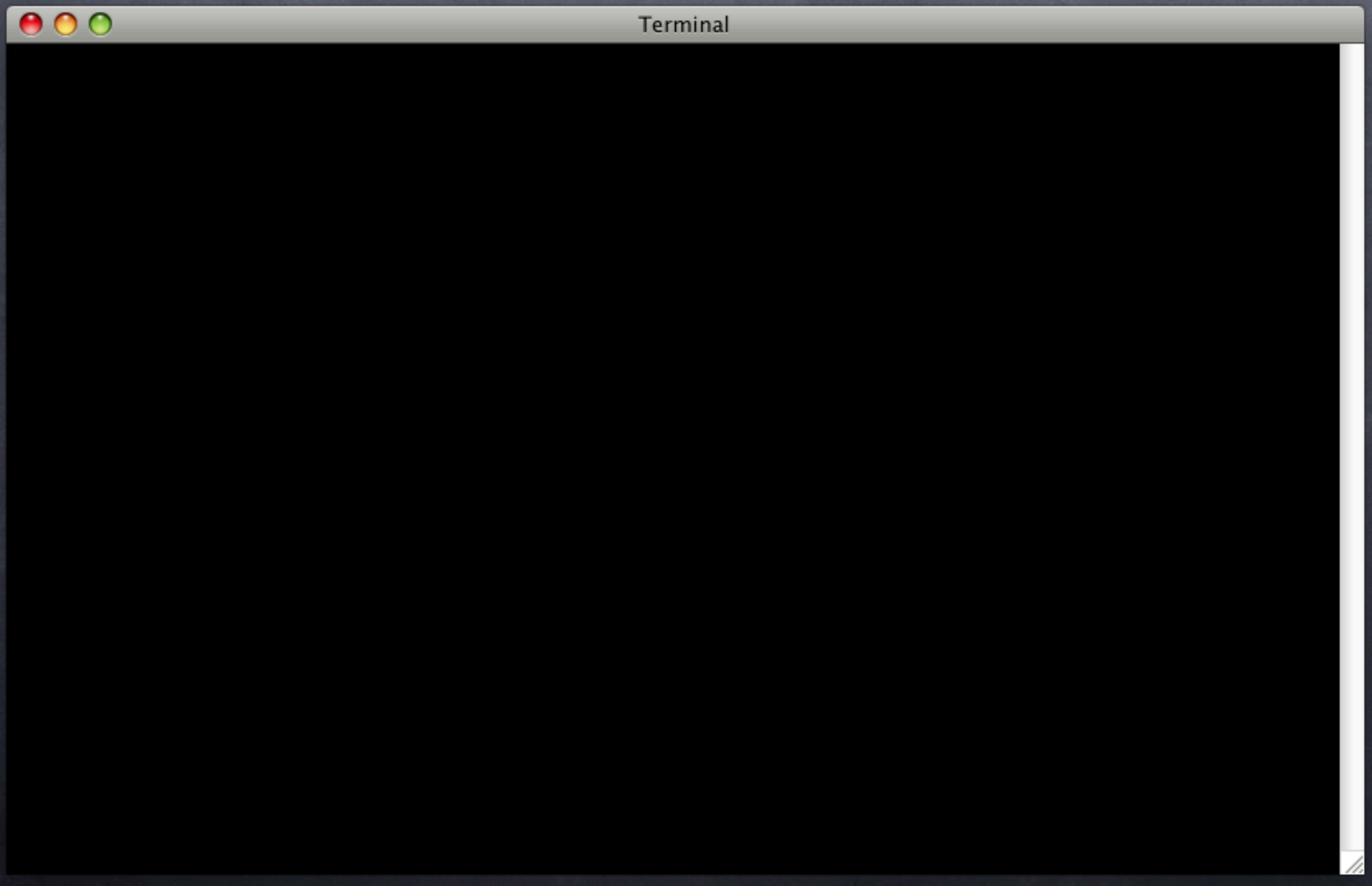
# Add Tests

Emacs

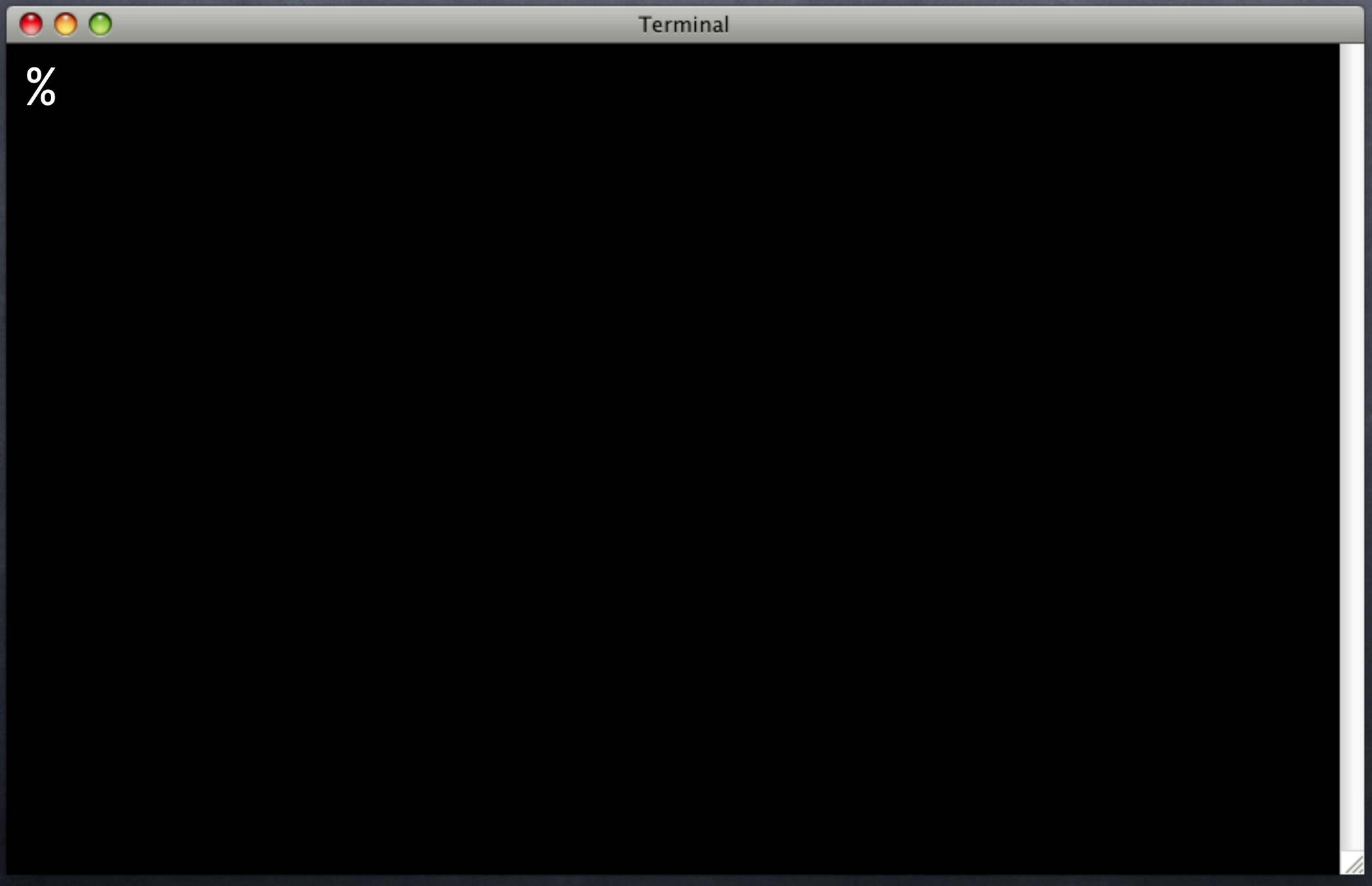
```
-- Insert test values.
INSERT INTO hs_tests VALUES
('a=>1,b=>2',  'a',   '1',   'a',   '{a,b}', '{1,2}'),
('a=>1',       'a',   '1',   'a',   '{a}',   '{1}'),
('foo=>bar',   'foo', 'bar', 'foo', '{foo}',  '{bar}'),
('1=>a,2=>b',  '1',   'a',   '1',   '{1,2}',  '{a,b}'),
(
    'name=>"Tom Lane",rank=>"H@xi3",sn=>ou812',
    'rank', 'H@xi3',
    'name',
    '{sn,name,rank}',
    '{ou812,"Tom Lane",H@xi3}'
);
```

--:--  try.sql       All  (SQL[ansi])----------

# More Repetition



# More Repetition



# More Repetition

Terminal

```
% pg_prove -d try hstore.sql
hstore.sql ..
All tests successful.
Files=1, Tests=20,  0 wallclock secs
( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

# More Repetition

Terminal

```
% pg_prove -d try hstore.sql
hstore.sql ..
All tests successful.
Files=1, Tests=20,  0 wallclock secs
( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

# More Repetition

Terminal

```
% pg_prove -d try hstore.sql
hstore.sql ..
All tests successful.
Files=1, Tests=20,  0 wallclock secs
( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

# Skipping Tests

# Skipping Tests

- Sometimes need to skip tests

# Skipping Tests

- Sometimes need to skip tests
  - Platform dependencies

# Skipping Tests

- Sometimes need to skip tests
  - Platform dependencies
  - Collation dependencies

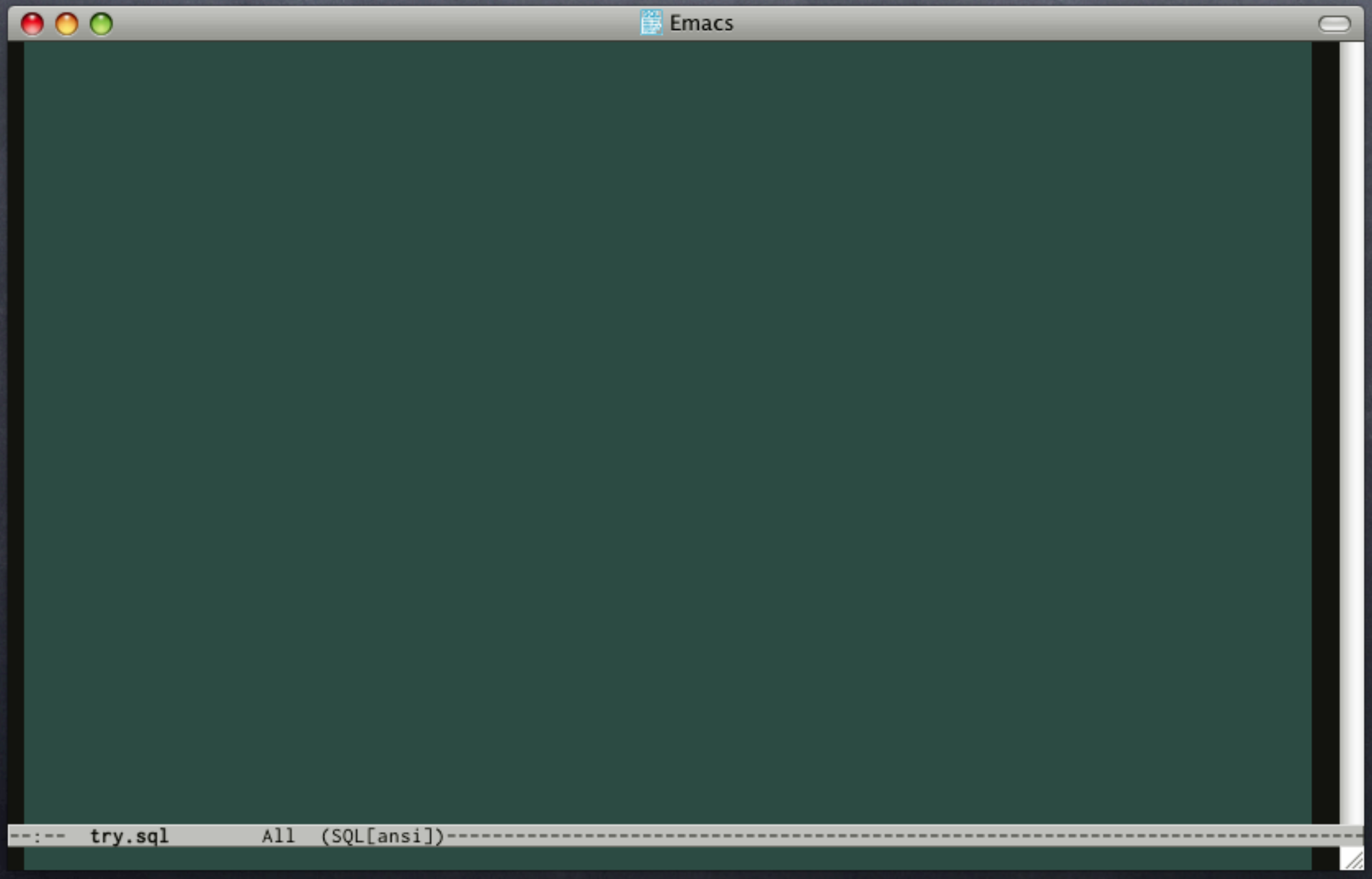
# Skipping Tests

- Sometimes need to skip tests
  - Platform dependencies
  - Collation dependencies
  - Version dependencies

# Skipping Tests

- Sometimes need to skip tests
  - Platform dependencies
  - Collation dependencies
  - Version dependencies
- Use skip()

# Skipping Tests



# Skipping Tests

Emacs

```
SELECT CASE os_name() WHEN 'darwin' THEN
   cmp_ok('Bjørn'::text, '>', 'Bjorn','ø > o')
ELSE
   skip('Collation-specific test')
END;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Skipping Tests

Emacs

```
SELECT CASE os_name() WHEN 'darwin' THEN
    cmp_ok('Bjørn'::text, '>', 'Bjorn','ø > o')
ELSE
    skip('Collation-specific test')
END;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Skipping Tests

Emacs

```
SELECT CASE os_name() WHEN 'darwin' THEN
    cmp_ok('Bjørn'::text, '>', 'Bjorn','ø > o')
ELSE
    skip('Collation-specific test')
END;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Skipping Tests

Emacs

```
SELECT CASE os_name() WHEN 'darwin' THEN
    cmp_ok('Bjørn'::text, '>', 'Bjorn','ø > o')
ELSE
    skip('Collation-specific test')
END;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

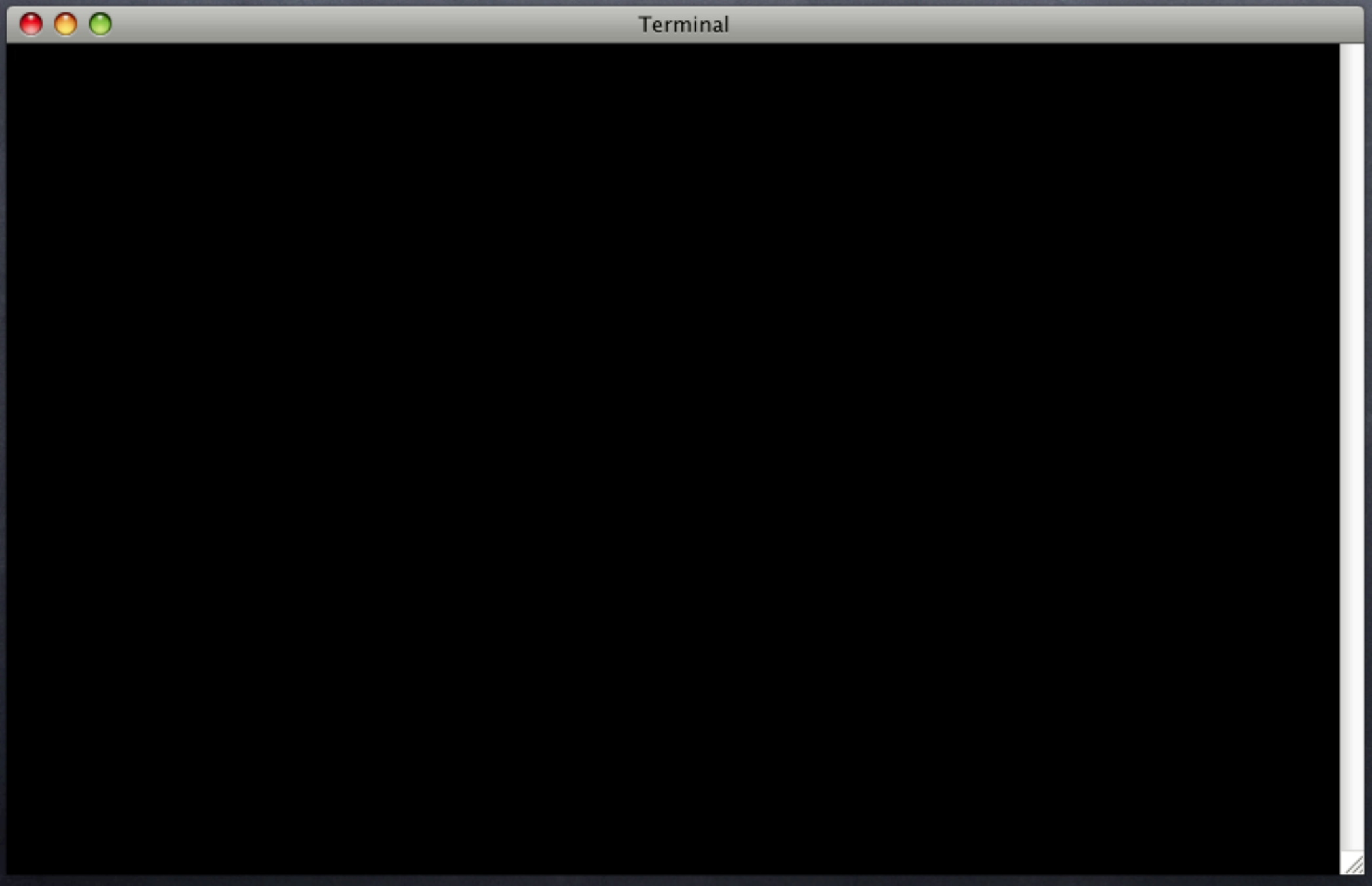
# Skipping Tests

Emacs

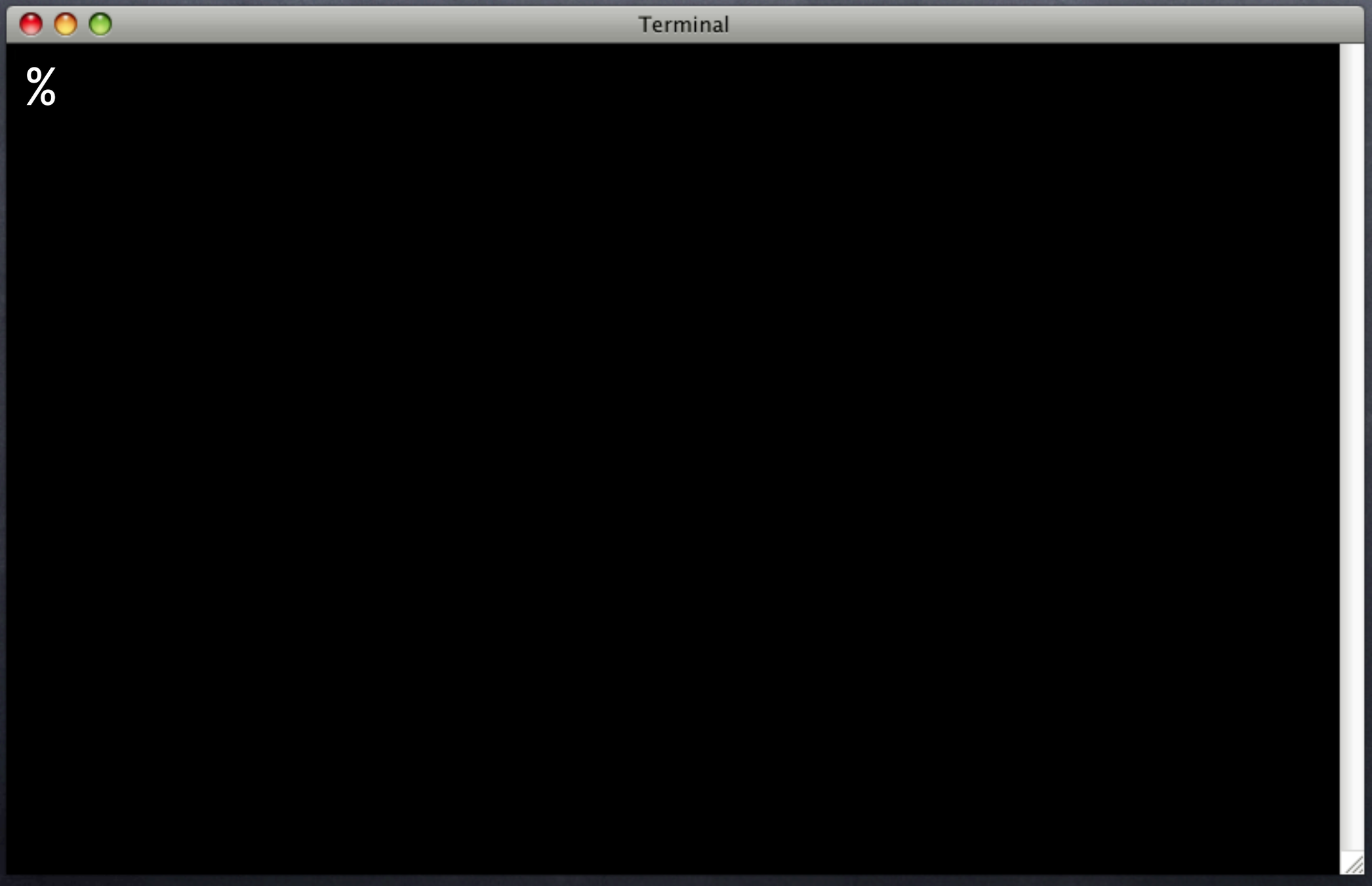
```
SELECT CASE os_name() WHEN 'darwin' THEN
    cmp_ok('Bjørn'::text, '>', 'Bjorn','ø > o')
ELSE
    skip('Collation-specific test')
END;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Skipping Tests



# Skipping Tests



# Skipping Tests

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..1
ok 1 - ø > o
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.01 sys =  0.03 CPU)
Result: PASS
```

# Skipping Tests

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..1
ok 1 - ø > o
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.01 sys =  0.03 CPU)
Result: PASS
```

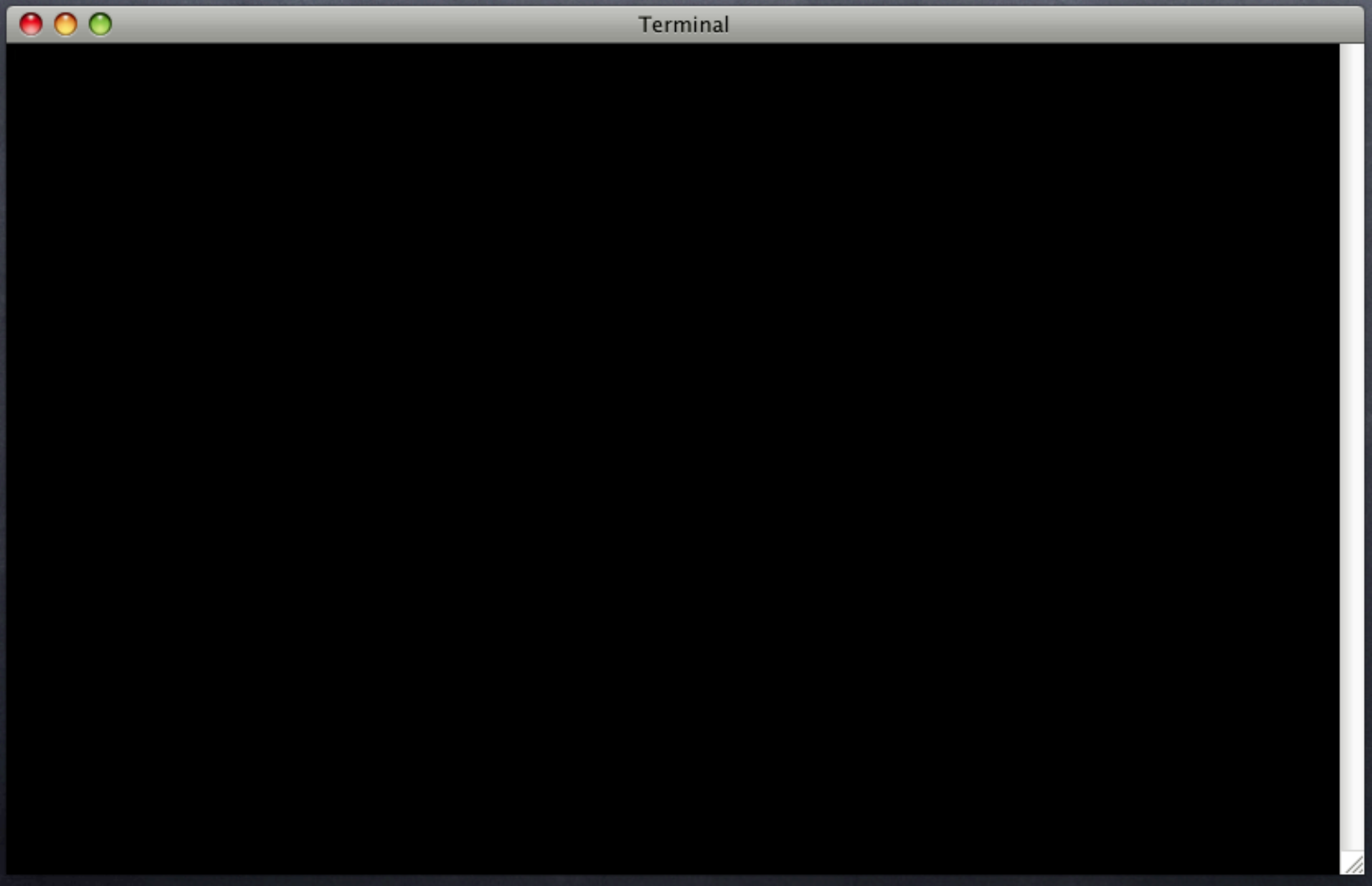
# Skipping Tests

Terminal

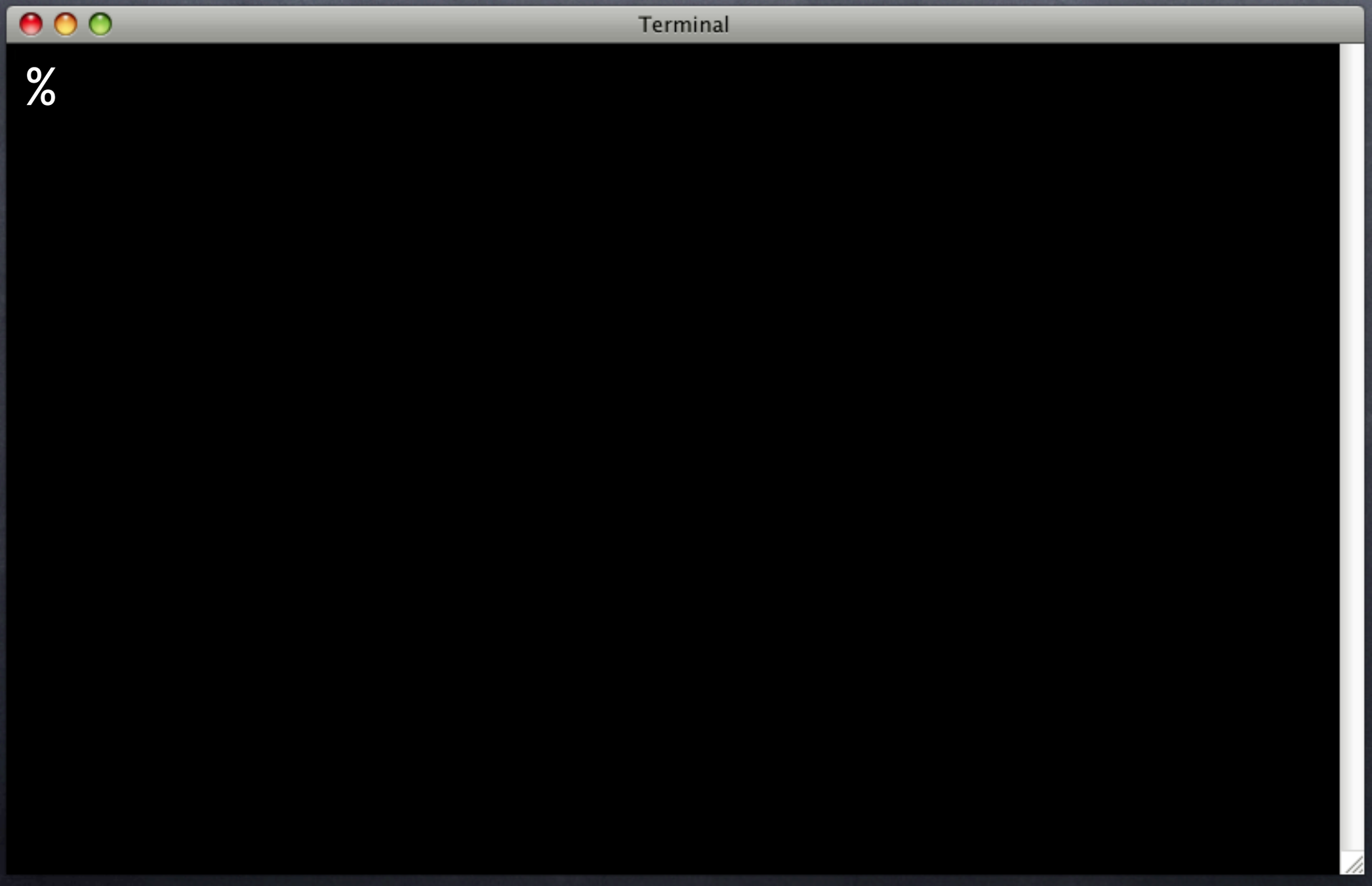
```
% pg_prove -v -d try collation.sql
collation.sql ..
1..1
ok 1 - ø > o
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.01 sys =  0.03 CPU)
Result: PASS
```

✔ Darwin

# Skipping Tests



# Skipping Tests



# Skipping Tests

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..1
ok 1 - SKIP: Collation-specific test
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

# Skipping Tests

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..1
ok 1 - SKIP: Collation-specific test
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

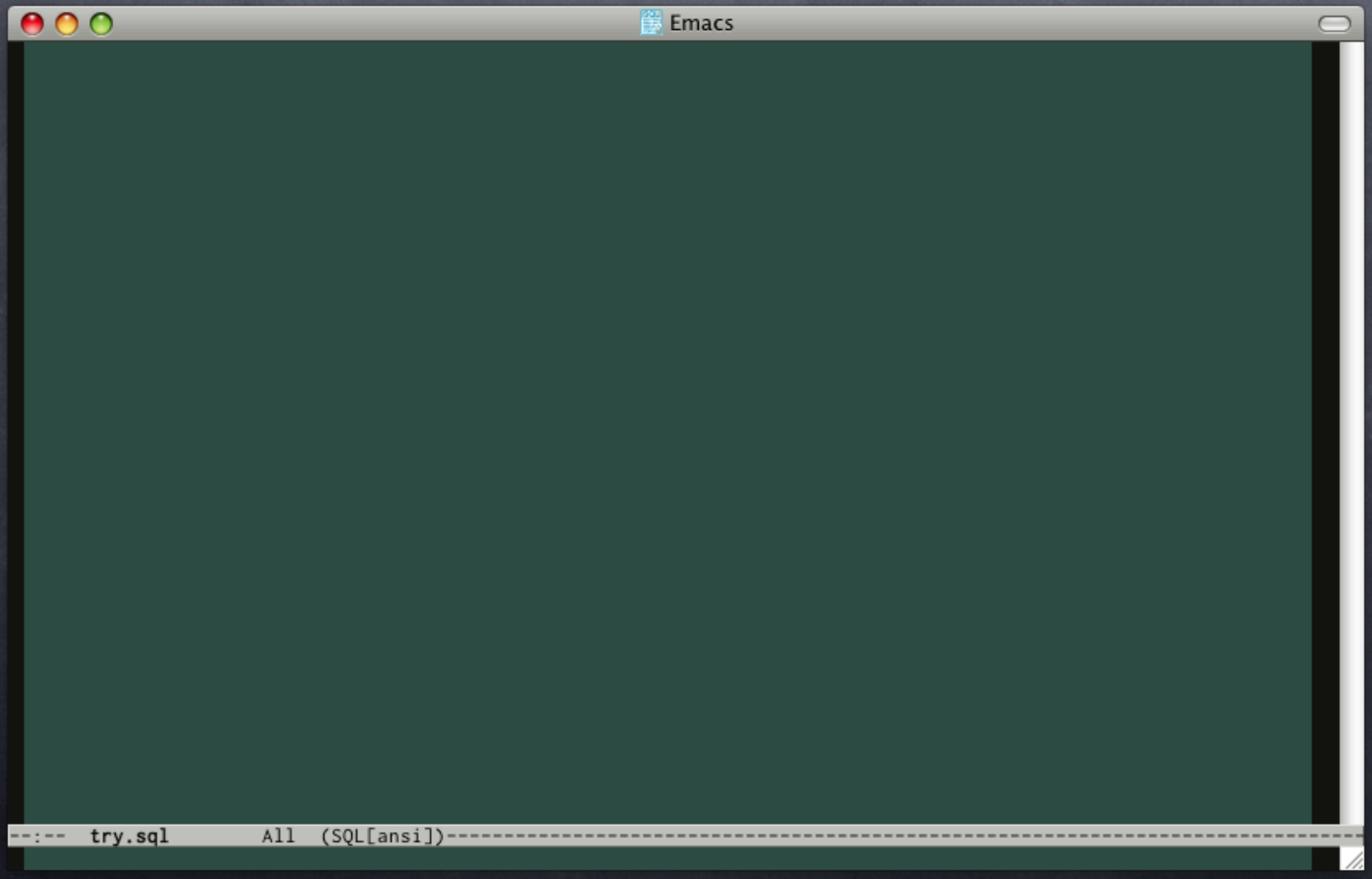
# Skipping Tests

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..1
ok 1 - SKIP: Collation-specific test
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
( 0.02 usr +  0.00 sys =  0.02 CPU)
Result: PASS
```

!Darwin

# Skip More



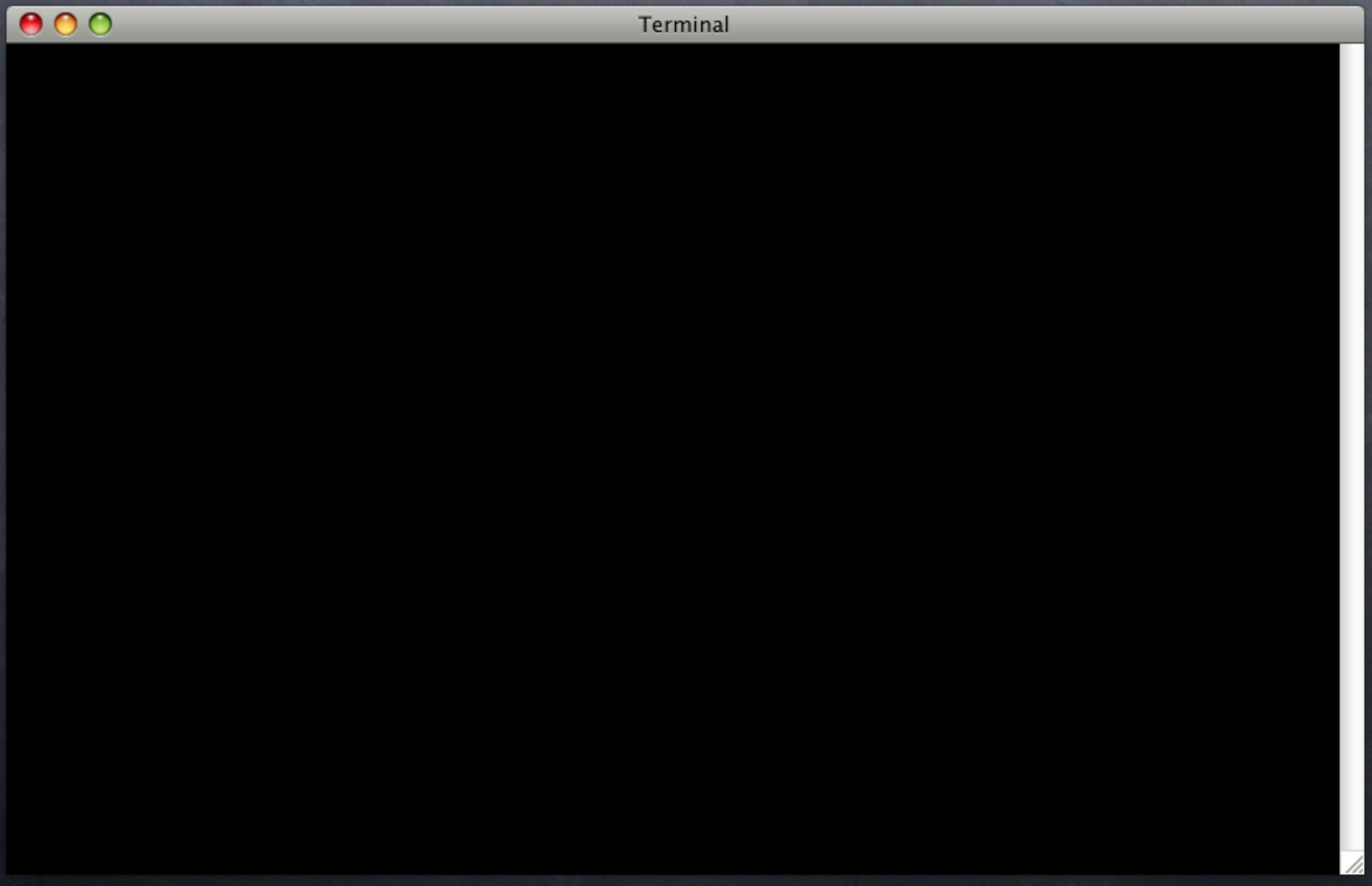
# Skip More

Emacs

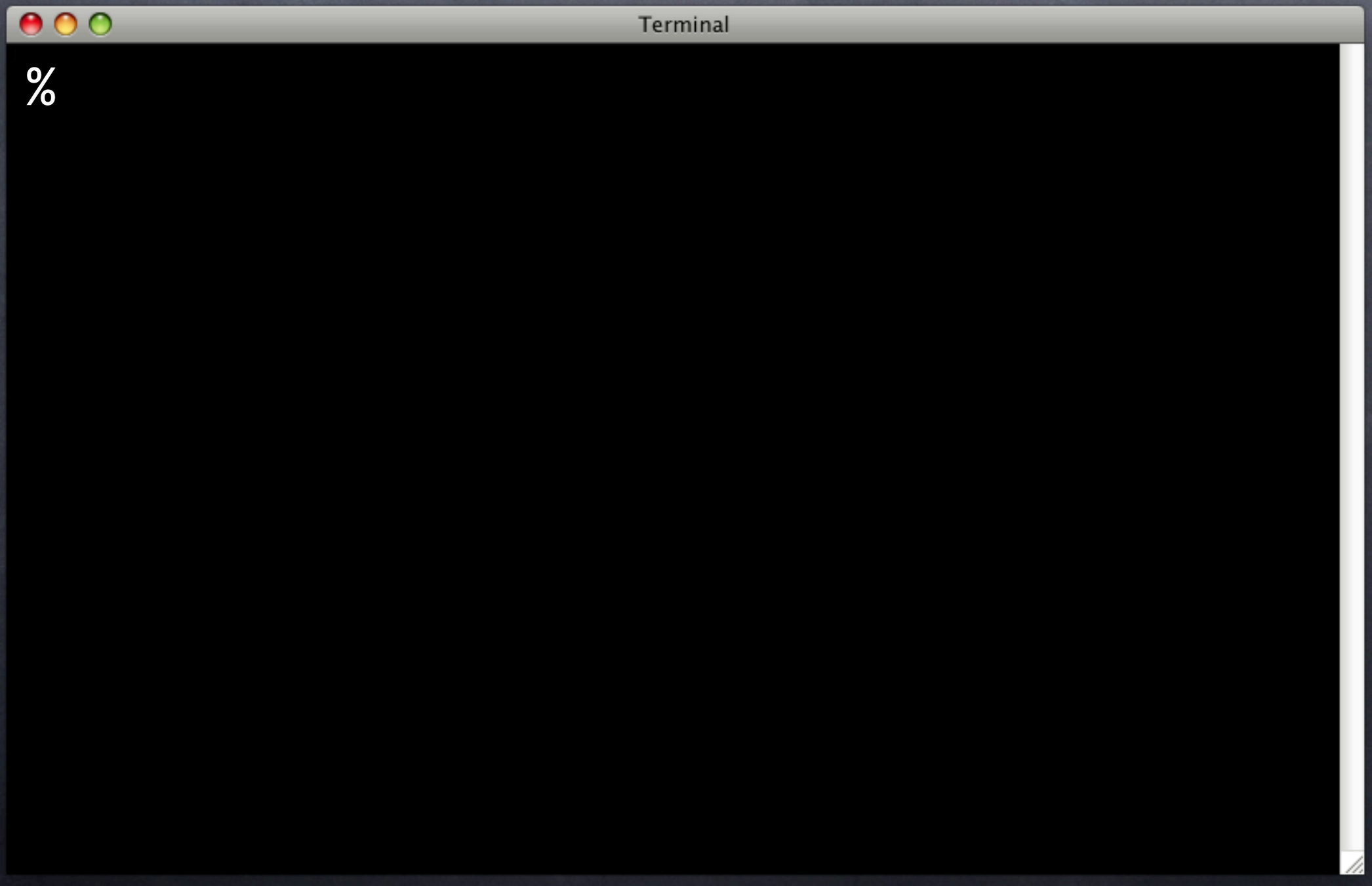
```
SELECT CASE os_name() WHEN 'darwin' THEN
    array_to_string( ARRAY[
        cmp_ok( 'Bjørn'::text, '>', 'Bjorn', 'ø > o' ),
        cmp_ok( 'Pınar'::text, '>', 'Pinar', 'ı > i' ),
        cmp_ok( 'José'::text,  '>', 'Jose',  'é > e' ),
        cmp_ok( 'Täp'::text,   '>', 'Tap',   'ä > a' )
    ], E'\n' )
ELSE
    skip('Collation-specific test', 4)
END;
```

--:--  try.sql  All  (SQL[ansi])----------

# Skip More



# Skip More



# Skip More

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..4
ok 1 - ø > o
ok 2 - ı > i
ok 3 - é > e
ok 4 - ä > a
ok
All tests successful.
```

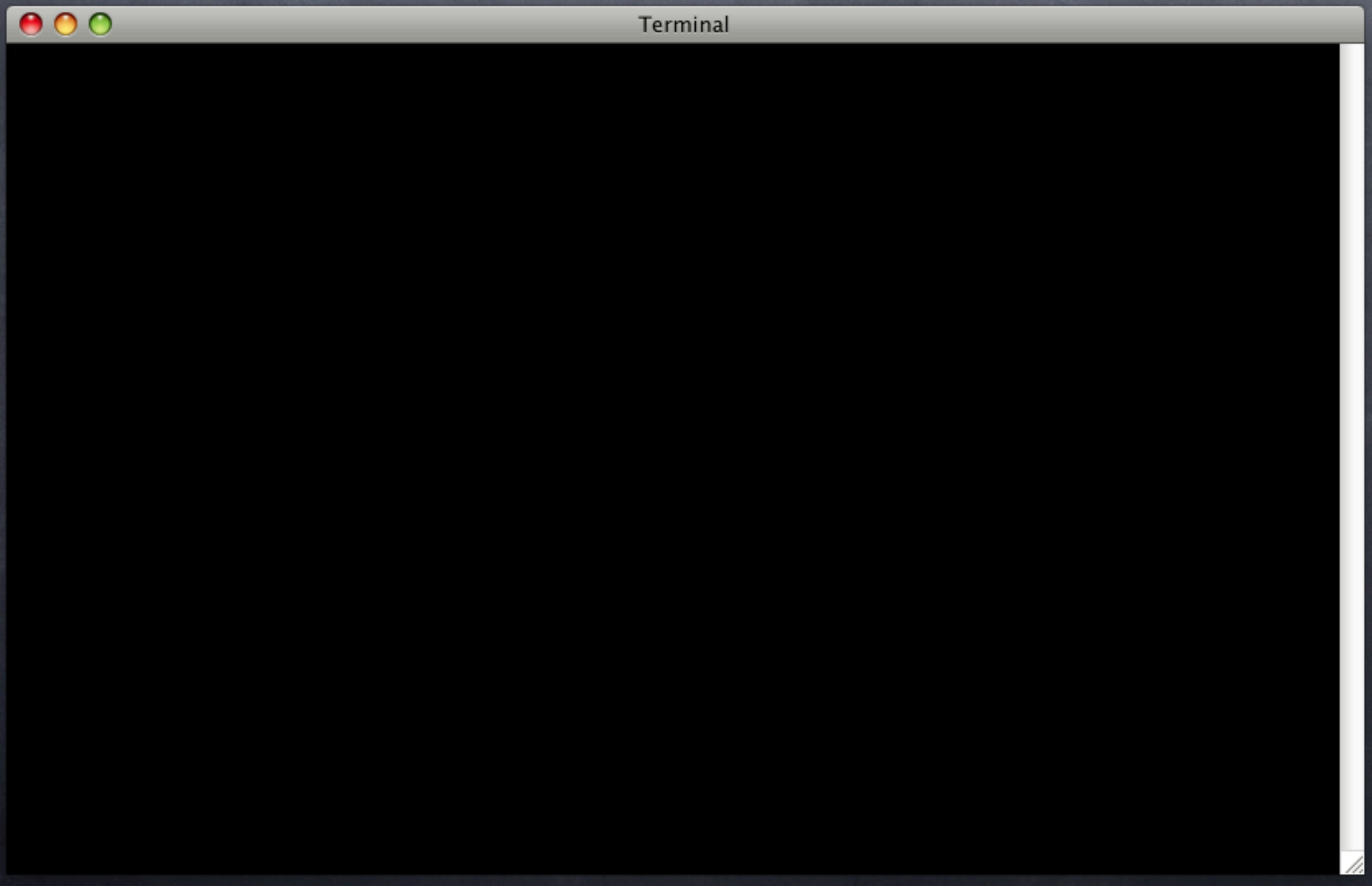
# Skip More

Terminal

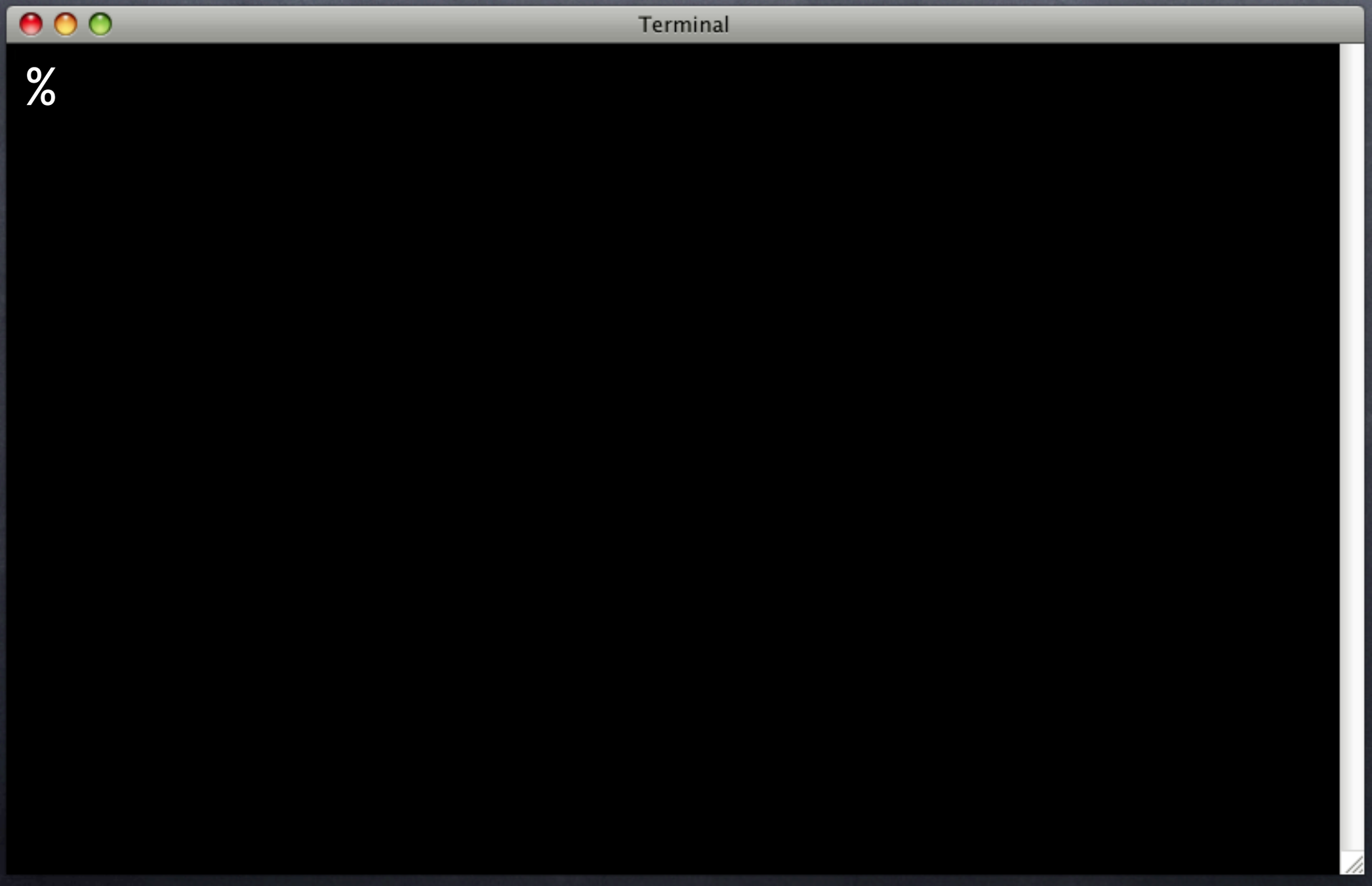
```
% pg_prove -v -d try collation.sql
collation.sql ..
1..4
ok 1 - ø > o
ok 2 - ı > i
ok 3 - é > e
ok 4 - ä > a
ok
All tests successful.
```

✔ Darwin

# Skip More



# Skip More



# Skip More

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..4
ok 1 - SKIP: Collation-specific test
ok 2 - SKIP: Collation-specific test
ok 3 - SKIP: Collation-specific test
ok 4 - SKIP: Collation-specific test
ok
All tests successful.
```

# Skip More

Terminal

```
% pg_prove -v -d try collation.sql
collation.sql ..
1..4
ok 1 - SKIP: Collation-specific test
ok 2 - SKIP: Collation-specific test
ok 3 - SKIP: Collation-specific test
ok 4 - SKIP: Collation-specific test
ok
All tests successful.
```

! Darwin

# Todo Tests

# Todo Tests

- Sometimes need to ignore failures

# Todo Tests

- Sometimes need to ignore failures
  - Features not implemented

# Todo Tests

- Sometimes need to ignore failures
  - Features not implemented
  - Unfixed regression

# Todo Tests

- Sometimes need to ignore failures
  - Features not implemented
  - Unfixed regression
  - Version dependences

# Todo Tests

- Sometimes need to ignore failures
  - Features not implemented
  - Unfixed regression
  - Version dependences
- Use todo()

# Todo Tests

# Todo Tests

Emacs

```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Todo Tests

Emacs

```
SELECT plan(4);

SELECT isa_ok('foo'::text,    'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT is( pg_typeof('foo'),    'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Todo Tests

Emacs

```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Todo Tests

Emacs

```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

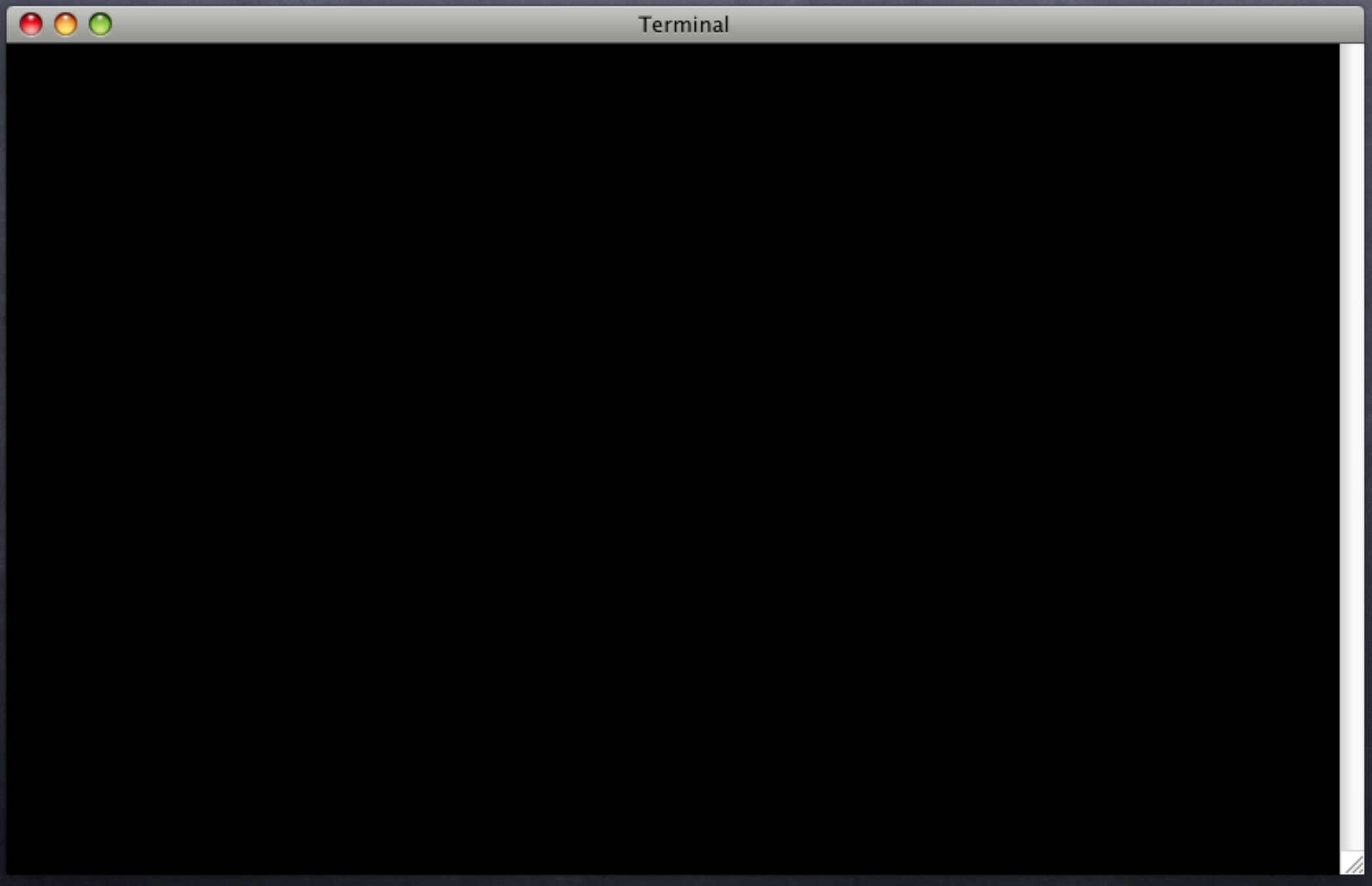
# Todo Tests

Emacs

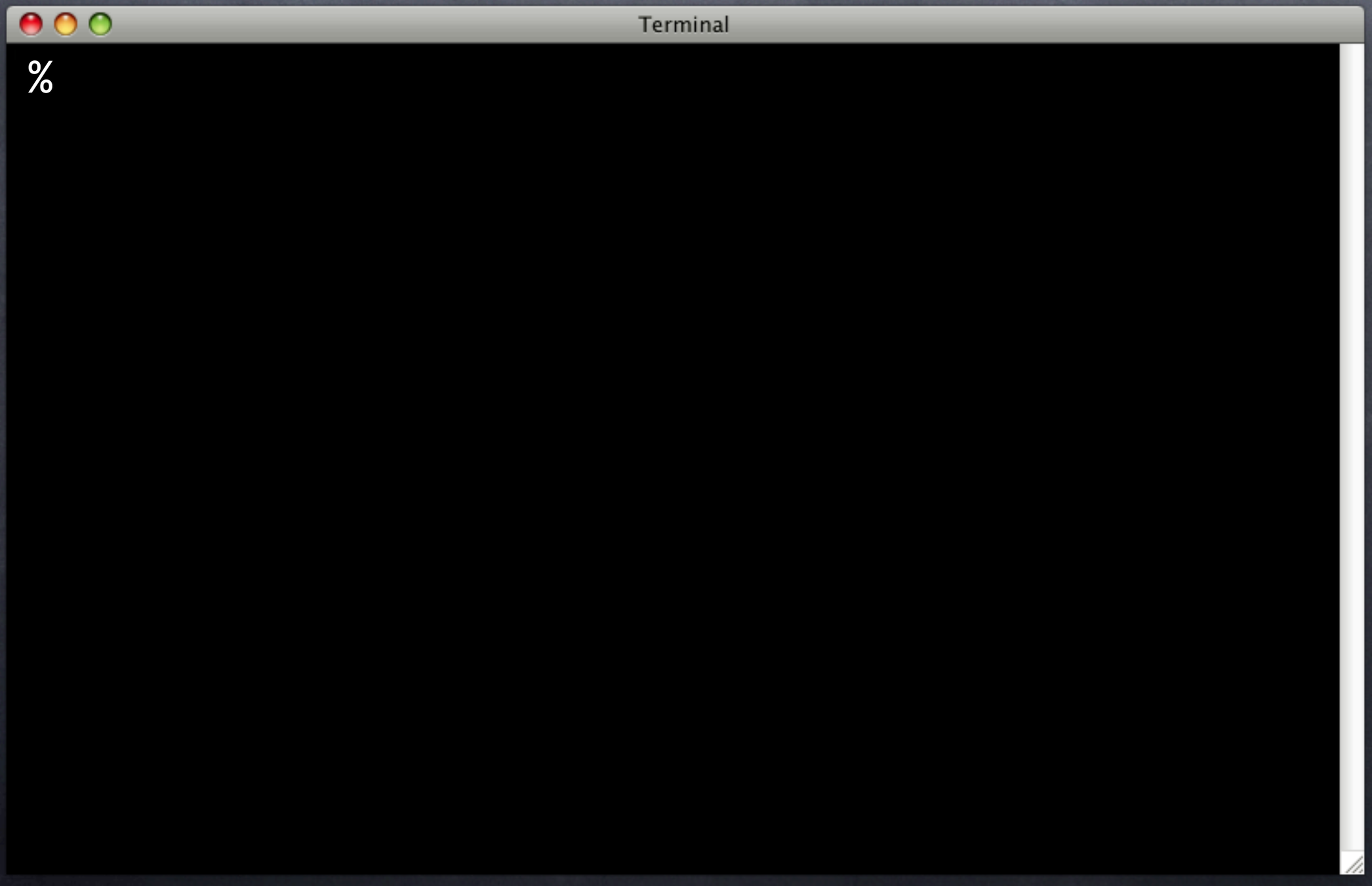
```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Todo Tests



# Todo Tests



# Todo Tests

Terminal

```
% pg_prove -v -d try text.sql
text.sql ..
ok 1 - text is text isa text
ok 2 - $$ is text isa text
not ok 3 - should be text
# Failed test 3: "should be text"
#         have: unknown
#         want: text
not ok 4 - $$ should be text
# Failed test 4: "$$ should be text"
#         have: unknown
#         want: text
# Looks like you failed 2 tests of 4
Failed 2/4 subtests
```

# Todo Tests

Terminal

```
% pg_prove -v -d try text.sql
text.sql ..
ok 1 - text is text isa text
ok 2 - $$ is text isa text
not ok 3 - should be text
# Failed test 3: "should be text"
#         have: unknown
#         want: text
not ok 4 - $$ should be text
# Failed test 4: "$$ should be text"
#         have: unknown
#         want: text
# Looks like you failed 2 tests of 4
Failed 2/4 subtests
```

Will get around to those…

# Todo Tests

Emacs

```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

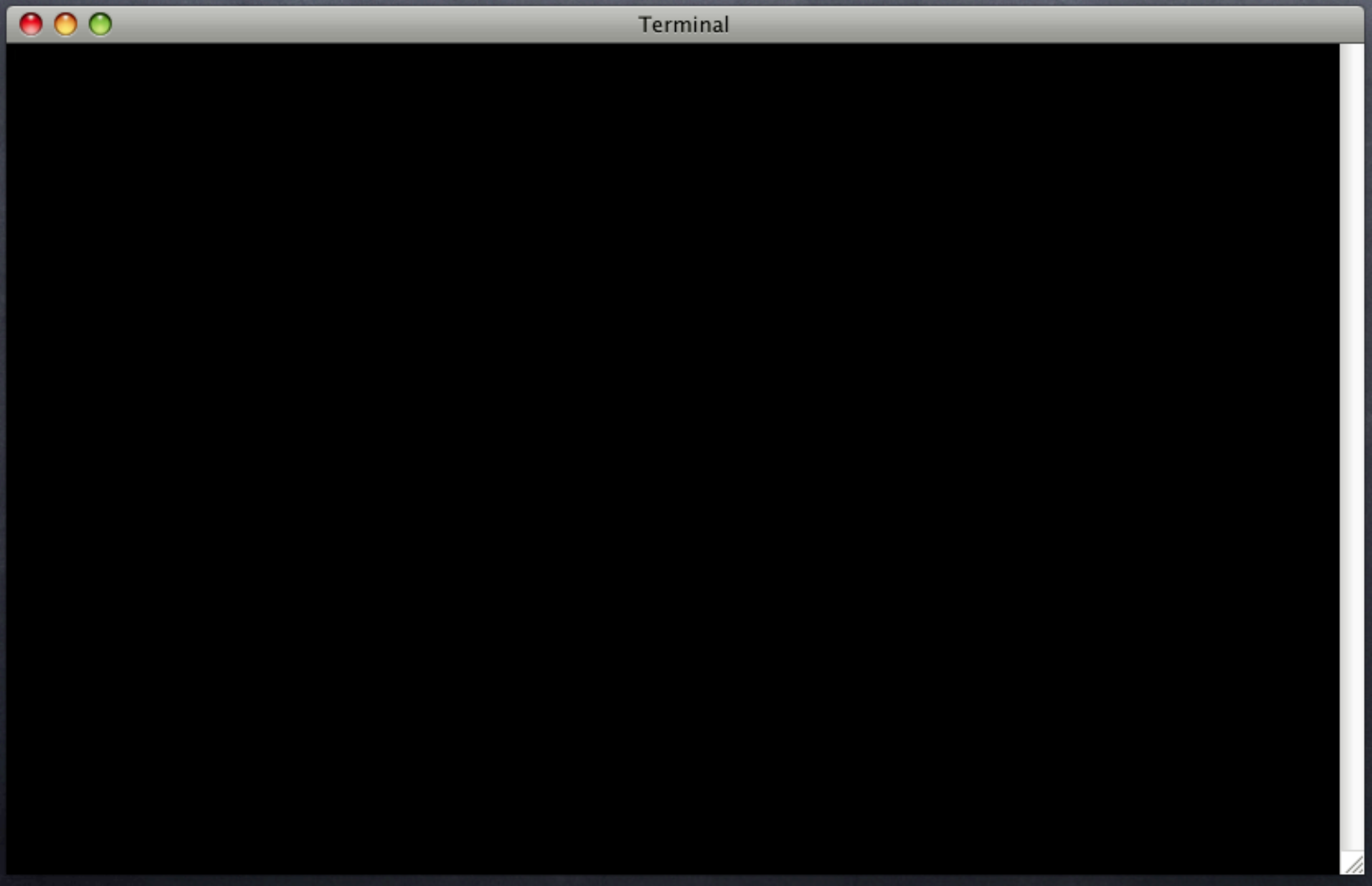
# Todo Tests

Emacs

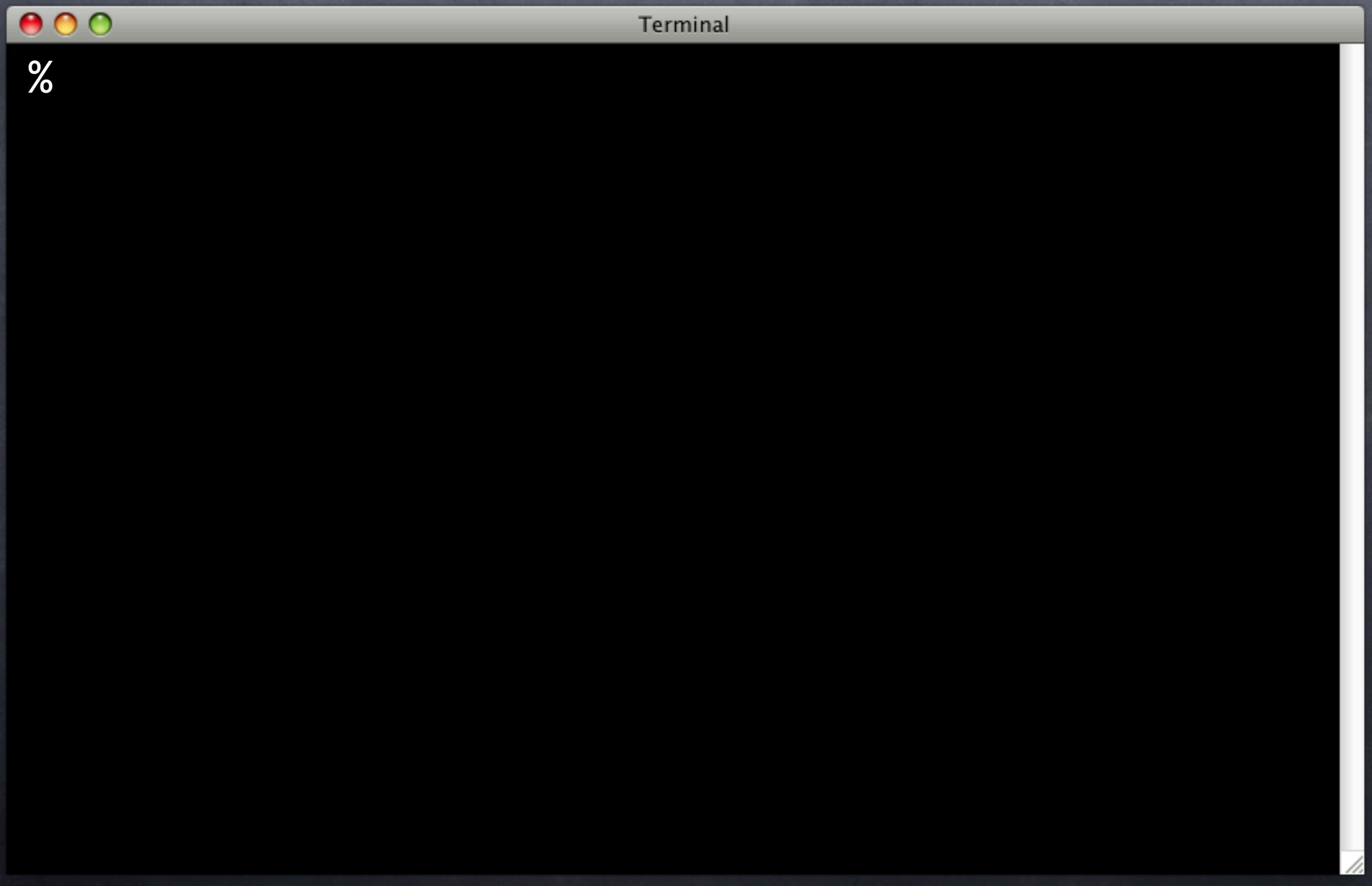
```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT * FROM todo('not written yet', 2 );
SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

--:--  try.sql  All  (SQL[ansi])----------

# Todo Tests



# Todo Tests



# Todo Tests

Terminal

```
% pg_prove -v -d try text.sql
text.sql ..
1..4
ok 1 - text is text isa text
ok 2 - $$ is text isa text
not ok 3 - should be text # TODO not written yet
# Failed (TODO) test 3: "should be text"
#          have: unknown
#          want: text
not ok 4 - $$ should be text # TODO not written yet
# Failed (TODO) test 4: "$$ should be text"
#          have: unknown
#          want: text
ok
All tests successful.
```

# Todo Tests

Terminal

```
% pg_prove -v -d try text.sql
text.sql ..
1..4
ok 1 - text is text isa text
ok 2 - $$ is text isa text
not ok 3 - should be text # TODO not written yet
# Failed (TODO) test 3: "should be text"
#         have: unknown
#         want: text
not ok 4 - $$ should be text # TODO not written yet
# Failed (TODO) test 4: "$$ should be text"
#         have: unknown
#         want: text
ok
All tests successful.
```

# Todo Tests

Terminal

```
% pg_prove -v -d try text.sql
text.sql ..
1..4
ok 1 - text is text isa text
ok 2 - $$ is text isa text
not ok 3 - should be text # TODO not written yet
# Failed (TODO) test 3: "should be text"
#         have: unknown
#         want: text
not ok 4 - $$ should be text # TODO not written yet
# Failed (TODO) test 4: "$$ should be text"
#         have: unknown
#         want: text
ok
All tests successful.
```

# Todo Tests

Emacs

```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT * FROM todo('not written yet', 2);
SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
```

--:--  try.sql  All  (SQL[ansi])----------

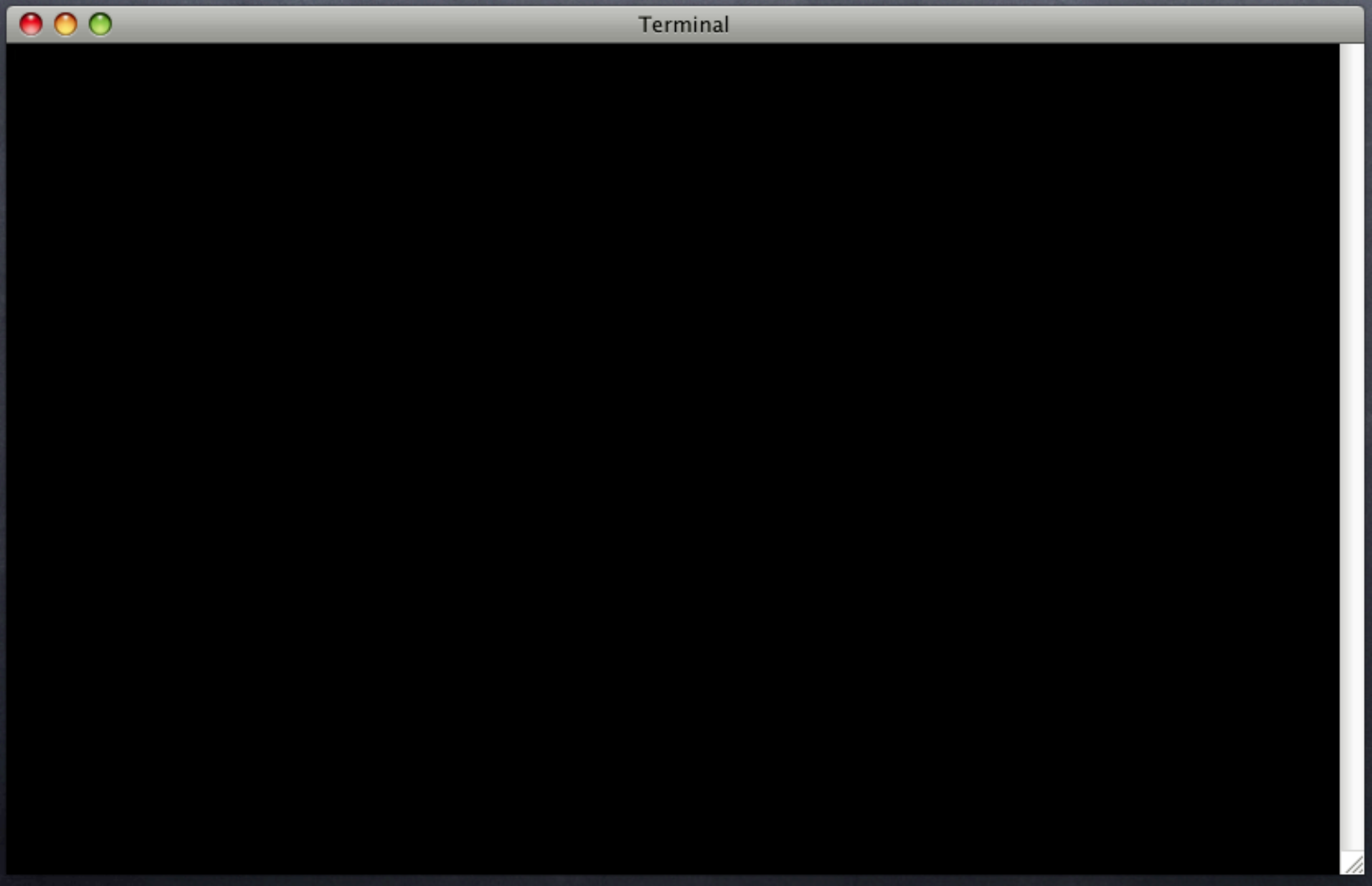
# Todo Tests

Emacs

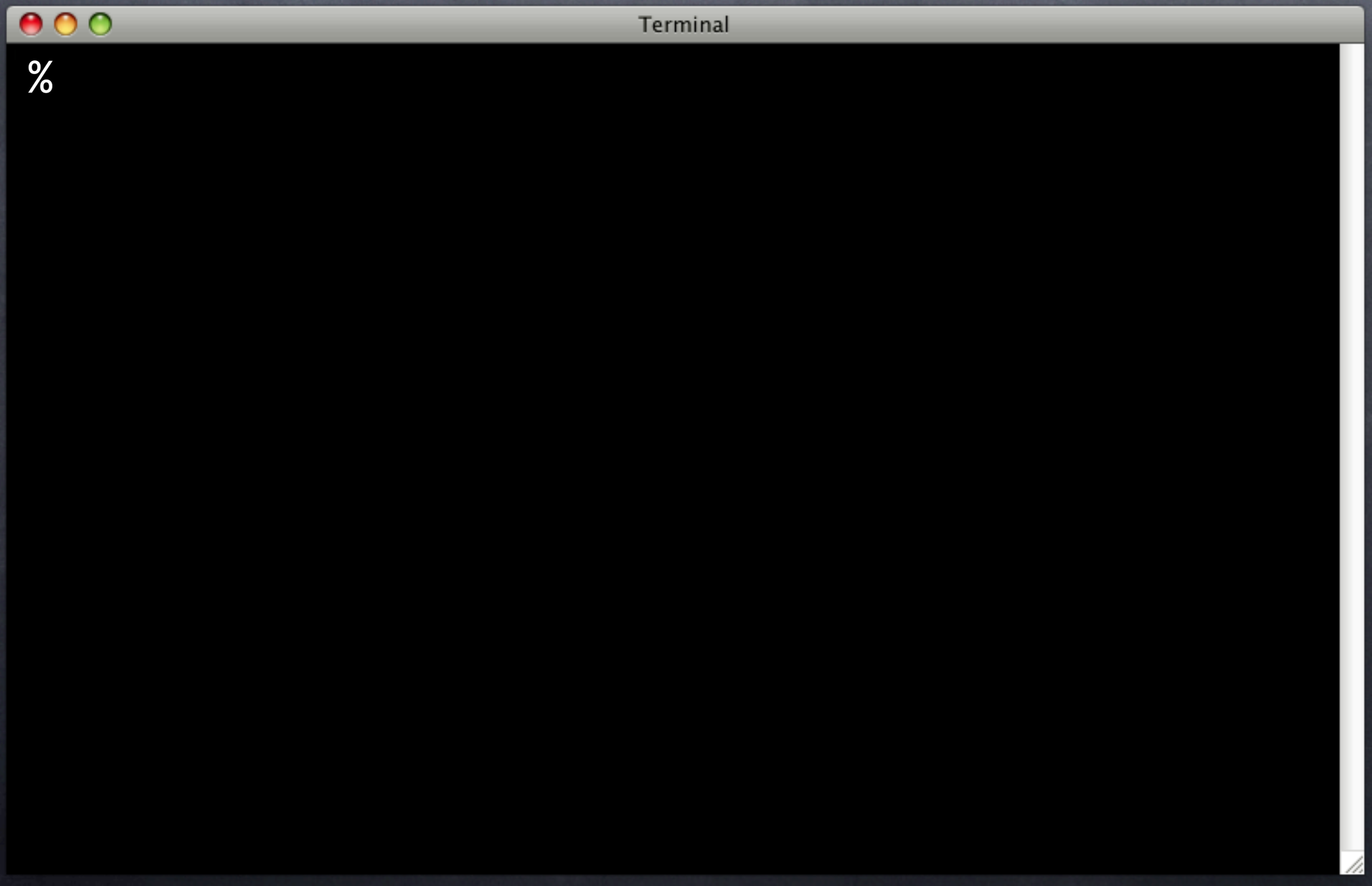
```
SELECT plan(4);

SELECT isa_ok('foo'::text,   'text', 'text is text');
SELECT isa_ok($$foo$$::text, 'text', '$$ is text');

SELECT * FROM todo_start('not written yet');
SELECT is( pg_typeof('foo'),   'text', 'should be text');
SELECT is( pg_typeof($$foo$$), 'text', '$$ should be text');
SELECT * FROM todo_end();
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Todo Tests



# Todo Tests



# Todo Tests

Terminal

```
% pg_prove -v -d try text.sql
text.sql ..
1..4
ok 1 - text is text isa text
ok 2 - $$ is text isa text
not ok 3 - should be text # TODO not written yet
# Failed (TODO) test 3: "should be text"
#          have: unknown
#          want: text
not ok 4 - $$ should be text # TODO not written yet
# Failed (TODO) test 4: "$$ should be text"
#          have: unknown
#          want: text
ok
All tests successful.
```

# Other Goodies

# Other Goodies

Emacs

```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Goodies

Emacs

```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Goodies

Emacs

```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Goodies

Emacs

```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Goodies

Emacs

```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Goodies

Emacs

```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

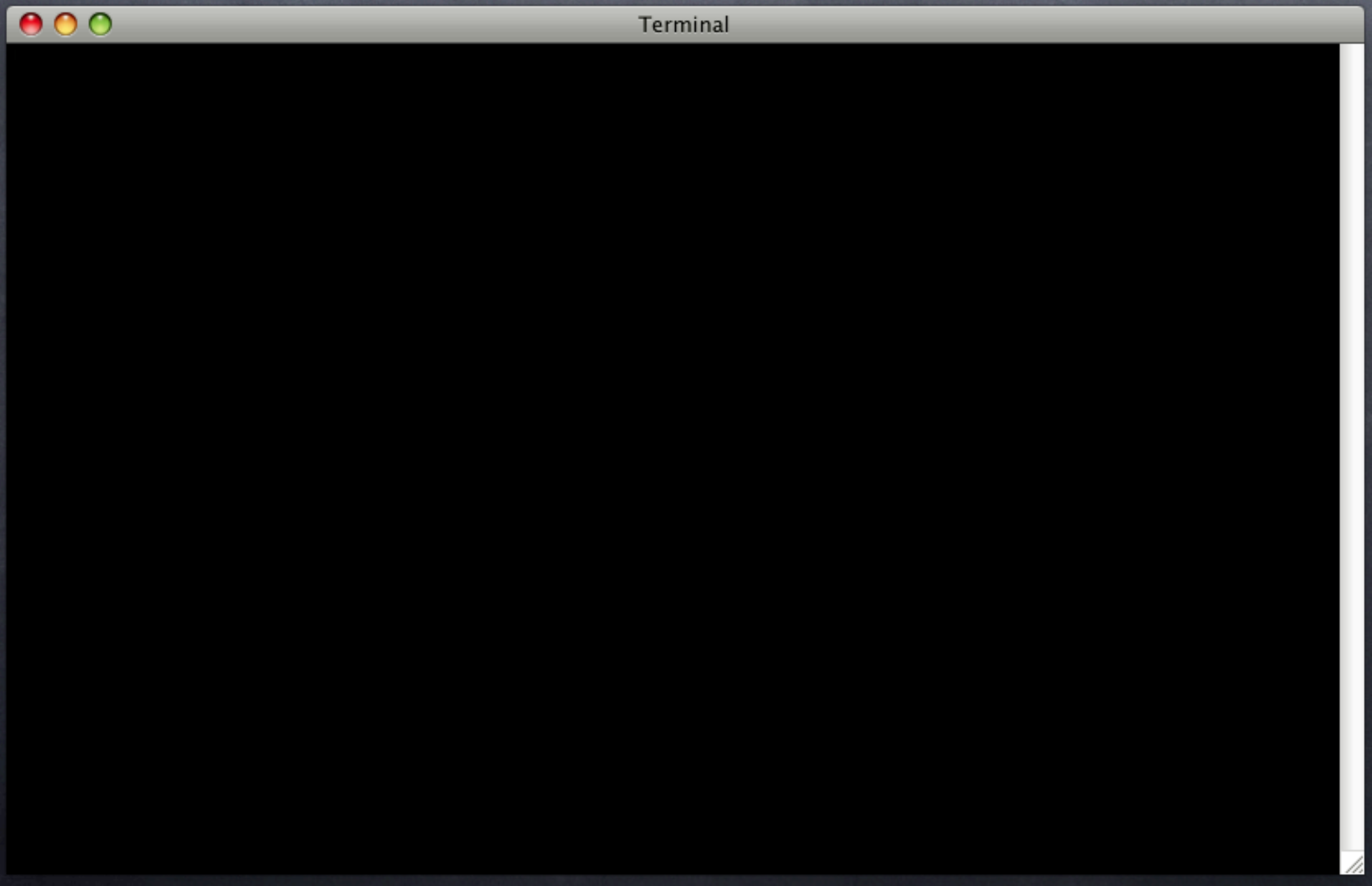
# Other Goodies

Emacs

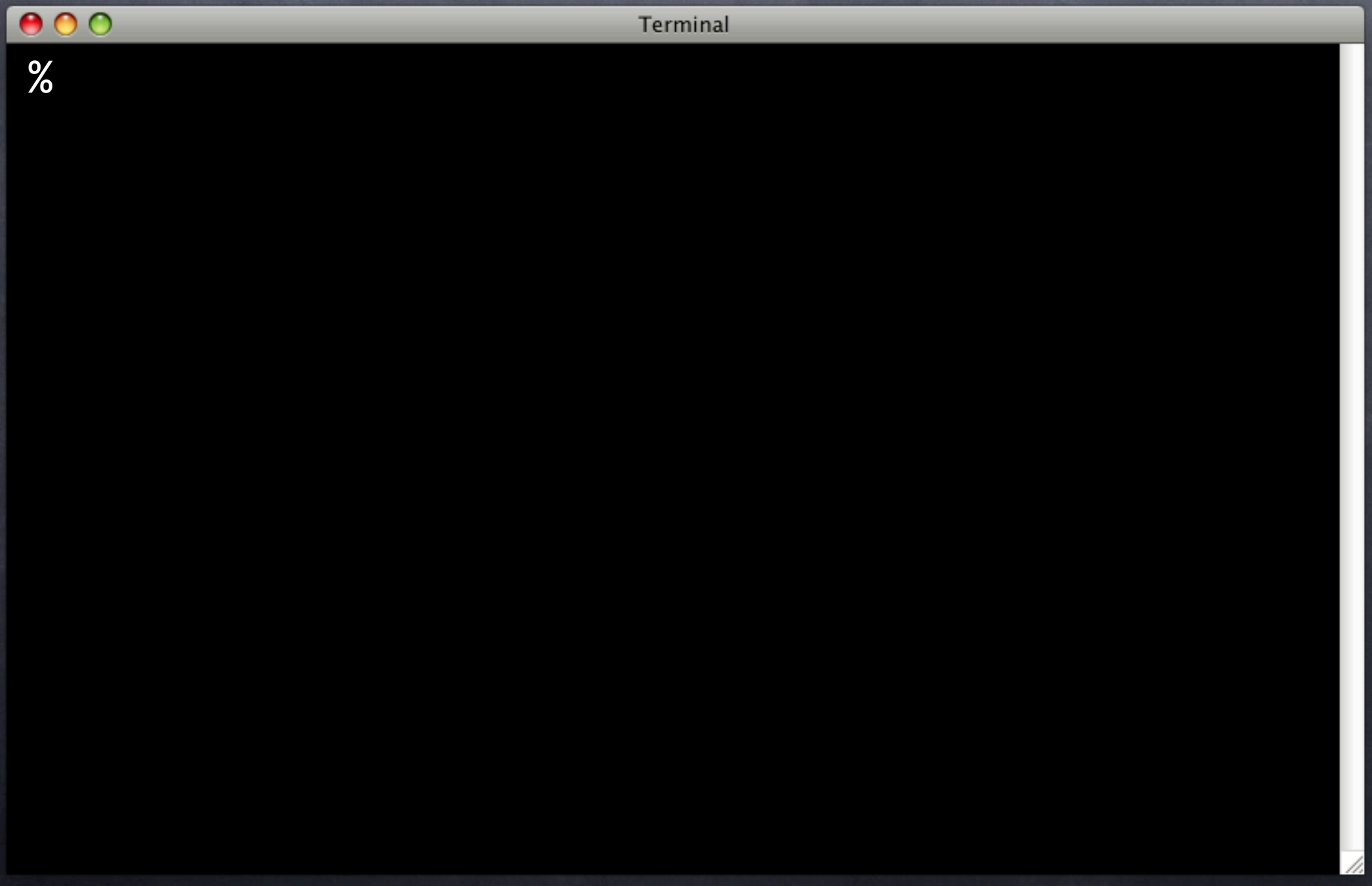
```
SELECT * FROM no_plan();

SELECT matches( :zip_code, '^[[:digit:]]{5}$' );
SELECT alike( :country, 'United %' );
SELECT throws_ok( 'SELECT lower(10.2)', '42883' );
SELECT throws_ok(
    'SELECT lower(10.2)',
    'function lower(numeric) does not exist'
);
SELECT lives_ok( 'SELECT 2 + 2', 'It''s alive!' );
SELECT diag( 'OMG TAP WTF?' );
SELECT performs_ok( 'SELECT fib(32)', 500 );

SELECT * FROM finish();
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Goodies



# Other Goodies



# Other Goodies

Terminal

```
% pg_prove -v -d try goodies.sql
goodies.sql ..
ok 1
ok 2
ok 3 - threw 42883
ok 4 - threw function lower(numeric) does not exist
ok 5 - It's alive!
# OMG TAP WTF?
ok 6 - Should run in less than 500 ms
1..6
ok
All tests successful.
```

# Other Goodies

Terminal

```
% pg_prove -v -d try goodies.sql
goodies.sql ..
ok 1
ok 2
ok 3 - threw 42883
ok 4 - threw function lower(numeric) does not exist
ok 5 - It's alive!
# OMG TAP WTF?
ok 6 - Should run in less than 500 ms
1..6
ok
All tests successful.
```

# Other Goodies

Terminal

```
% pg_prove -v -d try goodies.sql
goodies.sql ..
ok 1
ok 2
ok 3 - threw 42883
ok 4 - threw function lower(numeric) does not exist
ok 5 - It's alive!
# OMG TAP WTF?
ok 6 - Should run in less than 500 ms
1..6
ok
All tests successful.
```

# Other Goodies

Terminal

```
% pg_prove -v -d try goodies.sql
goodies.sql ..
ok 1
ok 2
ok 3 - threw 42883
ok 4 - threw function lower(numeric) does not exist
ok 5 - It's alive!
# OMG TAP WTF?
ok 6 - Should run in less than 500 ms
1..6
ok
All tests successful.
```

# Other Goodies

Terminal

```
% pg_prove -v -d try goodies.sql
goodies.sql ..
ok 1
ok 2
ok 3 - threw 42883
ok 4 - threw function lower(numeric) does not exist
ok 5 - It's alive!
# OMG TAP WTF?
ok 6 - Should run in less than 500 ms
1..6
ok
All tests successful.
```

# Pursuing your Query

# Pursuing your Query

- Several functions execute queries

# Pursuing your Query

- Several functions execute queries
- Seen throws_ok(), lives_ok(), performs_ok()

# Pursuing your Query

- Several functions execute queries
- Seen throws_ok(), lives_ok(), performs_ok()
- Take SQL statement argument

# Pursuing your Query

- Several functions execute queries
- Seen throws_ok(), lives_ok(), performs_ok()
- Take SQL statement argument
- PITA for complicated queries

# Pursuing your Query

- Several functions execute queries
- Seen throws_ok(), lives_ok(), performs_ok()
- Take SQL statement argument
- PITA for complicated queries
- Single quotes a particular PITA

# Pursuing your Query

- Several functions execute queries
- Seen throws_ok(), lives_ok(), performs_ok()
- Take SQL statement argument
- PITA for complicated queries
- Single quotes a particular PITA
- Alternative: Prepared Statements

# Pursuing your Query

# Pursuing your Query

Emacs

```
PREPARE new_user AS
INSERT INTO users (nick, pass, name)
VALUES ('theory', 's3kr1t', 'David Wheeler');

EXECUTE new_user;
SELECT throws_ok ('new_user');
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Pursuing your Query

Emacs

```
PREPARE new_user AS
INSERT INTO users (nick, pass, name)
VALUES ('theory', 's3kr1t', 'David Wheeler');

EXECUTE new_user;
SELECT throws_ok ('new_user');
```

--:--  try.sql  All  (SQL[ansi])----------

# Pursuing your Query

Emacs

```
PREPARE new_user AS
INSERT INTO users (nick, pass, name)
VALUES ('theory', 's3kr1t', 'David Wheeler');

EXECUTE new_user;
SELECT throws_ok ('new_user');
```

--:--  try.sql  All  (SQL[ansi])----------

# Pursuing your Query

Emacs

```
PREPARE new_user AS
INSERT INTO users (nick, pass, name)
VALUES ('theory', 's3kr1t', 'David Wheeler');

EXECUTE new_user;
SELECT throws_ok ('new_user');
```

--:--  try.sql        All  (SQL[ansi])----------

# Testing Relations

# Testing Relations

- Not everything is a scalar

# Testing Relations

- Not everything is a scalar
- It's a RELATIONAL database, after all

# Testing Relations

- Not everything is a scalar
- It's a RELATIONAL database, after all
- Only so much coercion to scalars to do

# Testing Relations

- Not everything is a scalar
- It's a RELATIONAL database, after all
- Only so much coercion to scalars to do
- Need to test

# Testing Relations

- Not everything is a scalar
- It's a RELATIONAL database, after all
- Only so much coercion to scalars to do
- Need to test
  - results

# Testing Relations

- Not everything is a scalar
- It's a RELATIONAL database, after all
- Only so much coercion to scalars to do
- Need to test
  - results
  - sets

# Testing Relations

- Not everything is a scalar
- It's a RELATIONAL database, after all
- Only so much coercion to scalars to do
- Need to test
  - results
  - sets
  - bags

# results_eq()

# results_eq()

- Tests query results

# results_eq()

- Tests query results
- Row-by-row comparison

# results_eq()

- Tests query results
- Row-by-row comparison
- In order

# results_eq()

- Tests query results
- Row-by-row comparison
- In order
- Test query and expected query may be:

# results_eq()

- Tests query results
- Row-by-row comparison
- In order
- Test query and expected query may be:
  - SQL statements

# results_eq()

- Tests query results
- Row-by-row comparison
- In order
- Test query and expected query may be:
  - SQL statements
  - Prepared statement names

# results_eq()

- Tests query results
- Row-by-row comparison
- In order
- Test query and expected query may be:
  - SQL statements
  - Prepared statement names
  - Cursor names

# Testing Results

# Testing Results

- Example: active_users()

# Testing Results

- Example: active_users()
- Returns set of active users

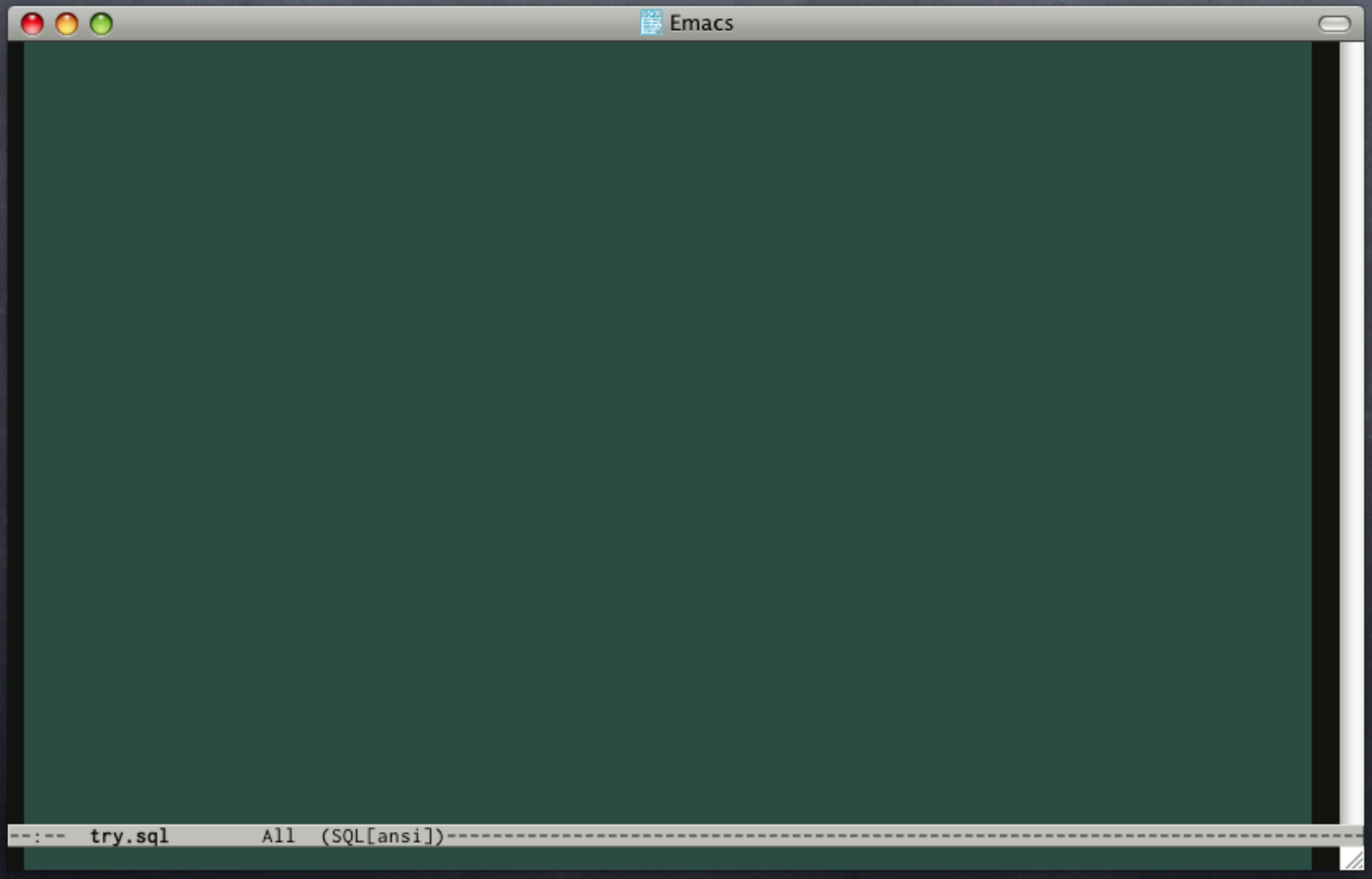
# Testing Results

- Example: active_users()
- Returns set of active users
- Test the function

# Testing Results

- Example: active_users()
- Returns set of active users
- Test the function
- Compare results

# results_eq() Arguments



# results_eq() Arguments

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active ORDER BY nick',
    'active_users() should return active users'
);
```

--:--  try.sql        All  (SQL[ansi])----------

# results_eq() Arguments

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active ORDER BY nick',
    'active_users() should return active users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

# results_eq() Arguments

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active ORDER BY nick',
    'active_users() should return active users'
);
```

--:--  try.sql        All  (SQL[ansi])----------

# results_eq() Arguments

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active ORDER BY nick',
    'active_users() should return active users'
);


SELECT results_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

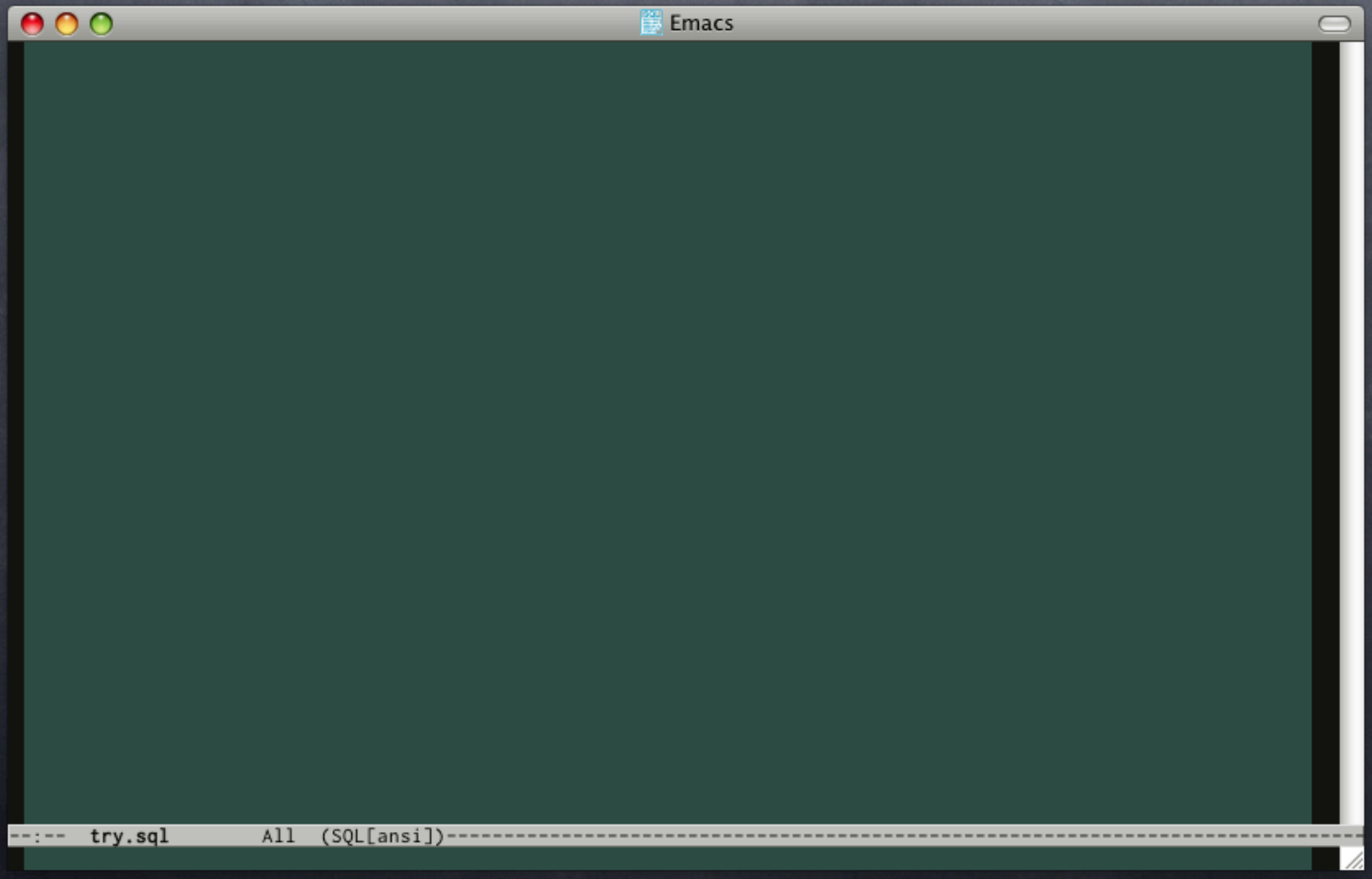
# results_eq() Arguments

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active ORDER BY nick',
    'active_users() should return active users'
);


SELECT results_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

# results_eq() Arguments



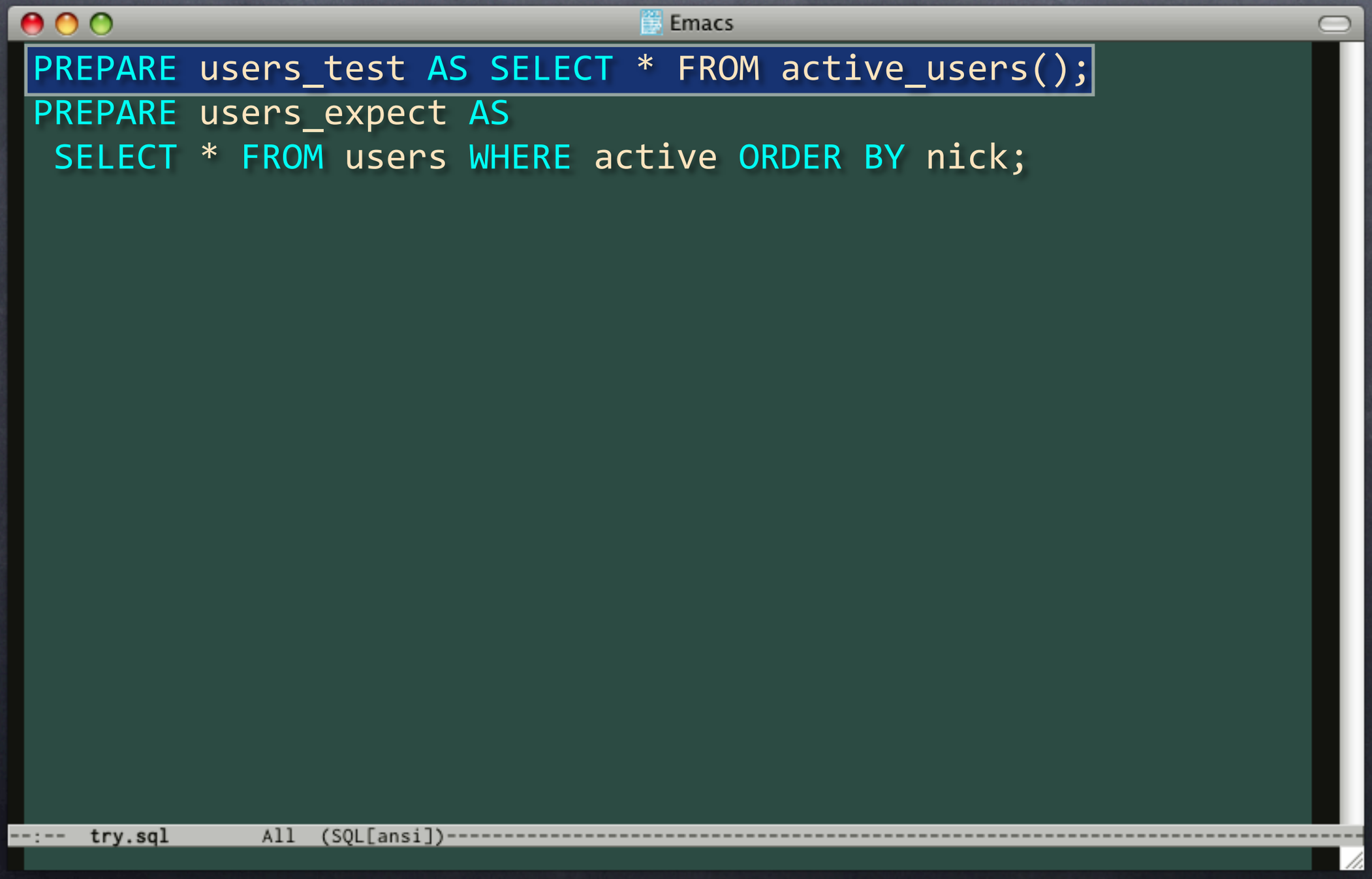
# results_eq() Arguments

Emacs

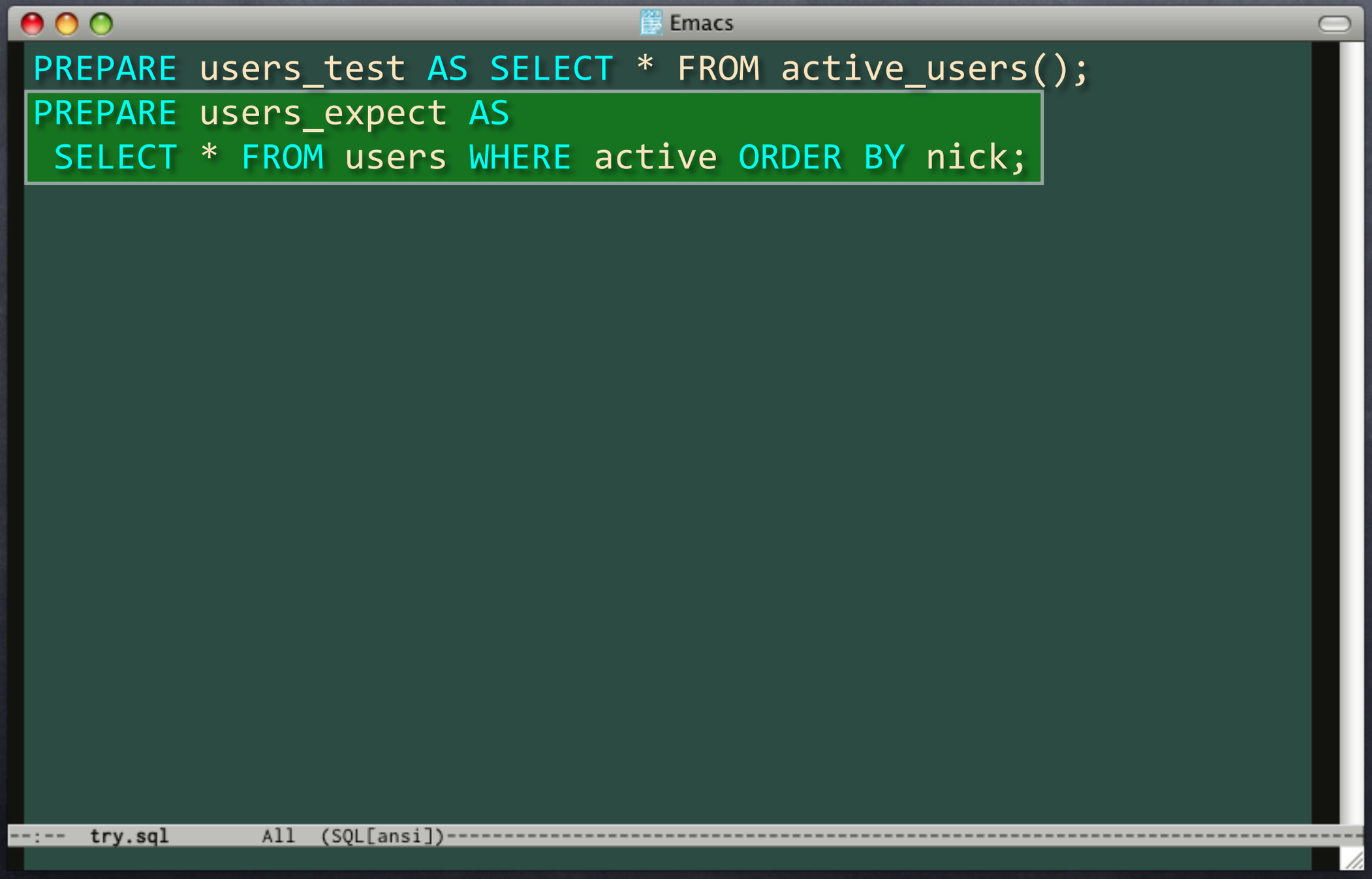
```
PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active ORDER BY nick;
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# results_eq() Arguments



# results_eq() Arguments



# results_eq() Arguments

Emacs

```
PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active ORDER BY nick;


SELECT results_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

# results_eq() Arguments

Emacs

```
PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active ORDER BY nick;


SELECT results_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql        All  (SQL[ansi])----------

Emacs

--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;

SELECT results_eq(
    'EXECUTE get_users_test(true)',
    'EXECUTE get_users_expect(true)',
    'We should have active users'
);
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;

SELECT results_eq(
    'EXECUTE get_users_test(true)',
    'EXECUTE get_users_expect(true)',
    'We should have active users'
);
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;

SELECT results_eq(
    'EXECUTE get_users_test(true)',
    'EXECUTE get_users_expect(true)',
    'We should have active users'
);
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;

SELECT results_eq(
    'EXECUTE get_users_test(true)',
    'EXECUTE get_users_expect(true)',
    'We should have active users'
);

SELECT results_eq(
    'EXECUTE get_users_test(false)',
    'EXECUTE get_users_expect(false)',
    'We should have inactive users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;

SELECT results_eq(
    'EXECUTE get_users_test(true)',
    'EXECUTE get_users_expect(true)',
    'We should have active users'
);

SELECT results_eq(
    'EXECUTE get_users_test(false)',
    'EXECUTE get_users_expect(false)',
    'We should have inactive users'
);
```

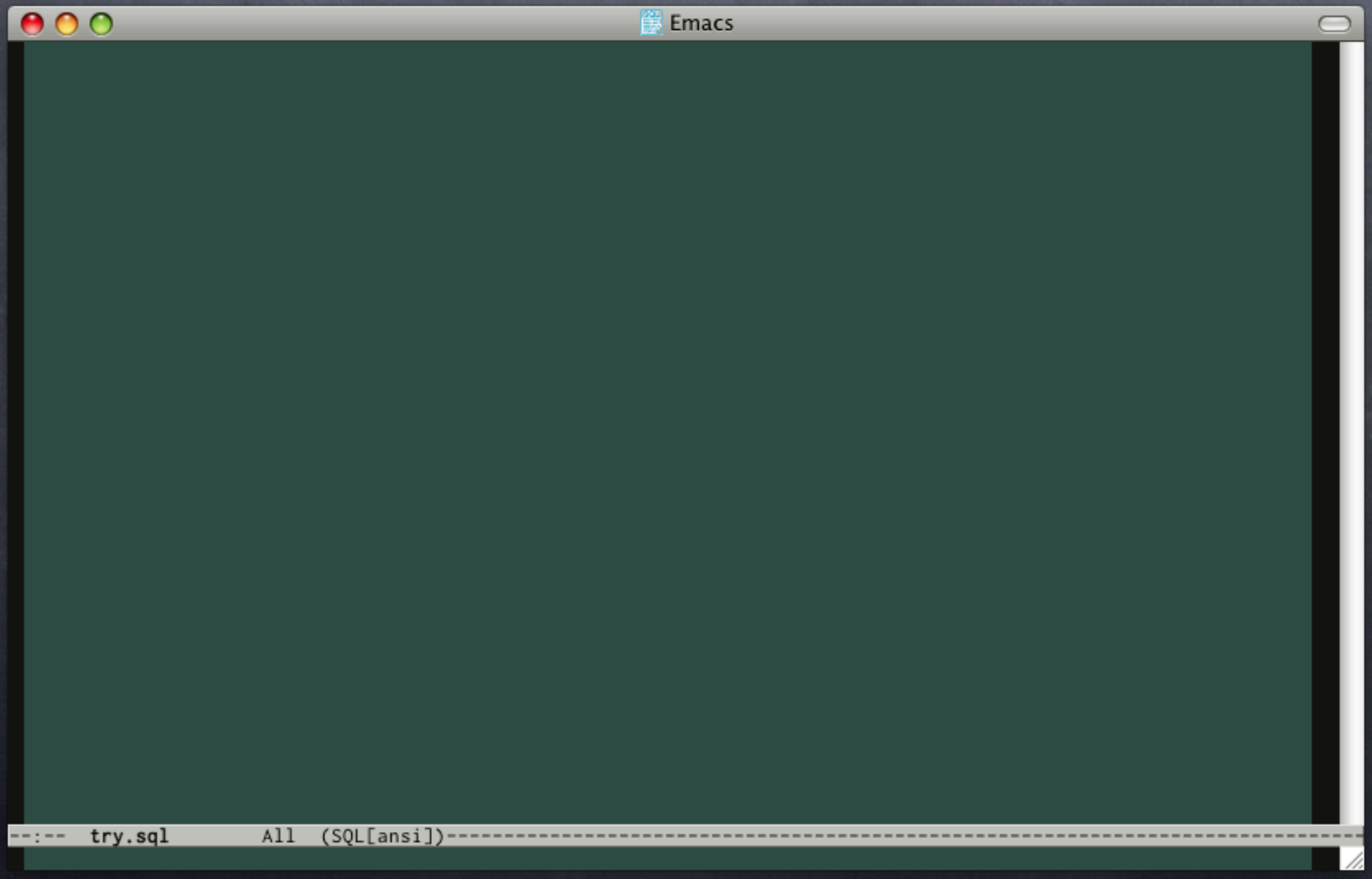
```
--:--  try.sql       All  (SQL[ansi])----------
```

Emacs

```
PREPARE get_users_test(boolean) AS
 SELECT * FROM get_users($1);

PREPARE get_users_expect(boolean) AS
 SELECT * FROM users
  WHERE active = $1
  ORDER BY nick;

SELECT results_eq(
    'EXECUTE get_users_test(true)',
    'EXECUTE get_users_expect(true)',
    'We should have active users'
);

SELECT results_eq(
    'EXECUTE get_users_test(false)',
    'EXECUTE get_users_expect(false)',
    'We should have inactive users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

# results_eq() Single Column & Array



# results_eq() Single Column & Array

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_user_nicks()',
    ARRAY[ 'anna', 'strongrrl', 'theory' ],
    'Should have active nicks'
);
```

--:--  try.sql        All  (SQL[ansi])----------

# results_eq() Single Column & Array

Emacs

```
SELECT results_eq(
    'SELECT * FROM active_user_nicks()',
    ARRAY[ 'anna', 'strongrrl', 'theory' ],
    'Should have active nicks'
);
```

--:--  try.sql        All  (SQL[ansi])----------

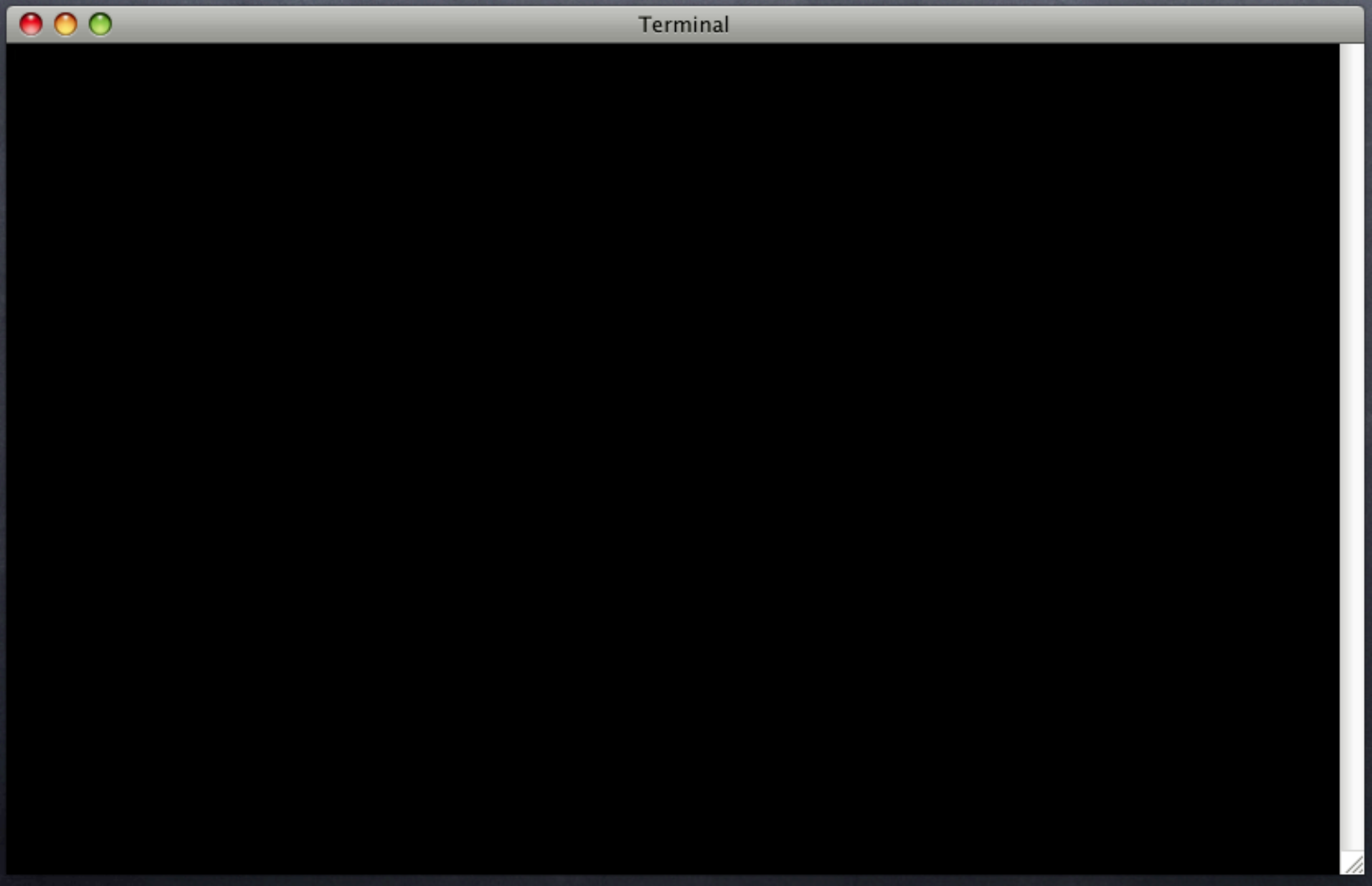
# results_eq() Single Column & Array

Emacs

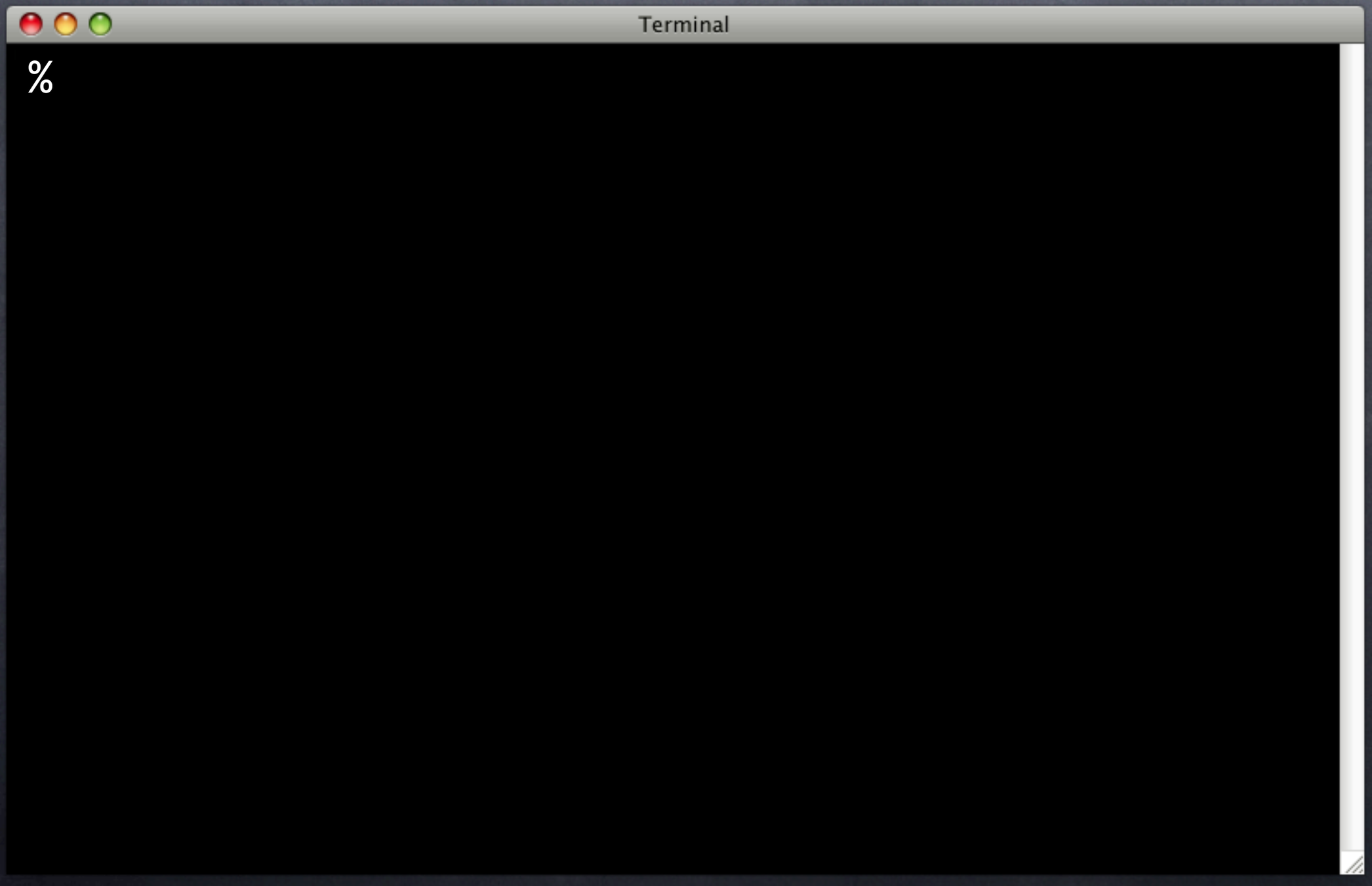
```
SELECT results_eq(
    'SELECT * FROM active_user_nicks()',
    ARRAY[ 'anna', 'strongrrl', 'theory' ],
    'Should have active nicks'
);
```

--:--  try.sql        All  (SQL[ansi])----------

# Testing Results



# Testing Results

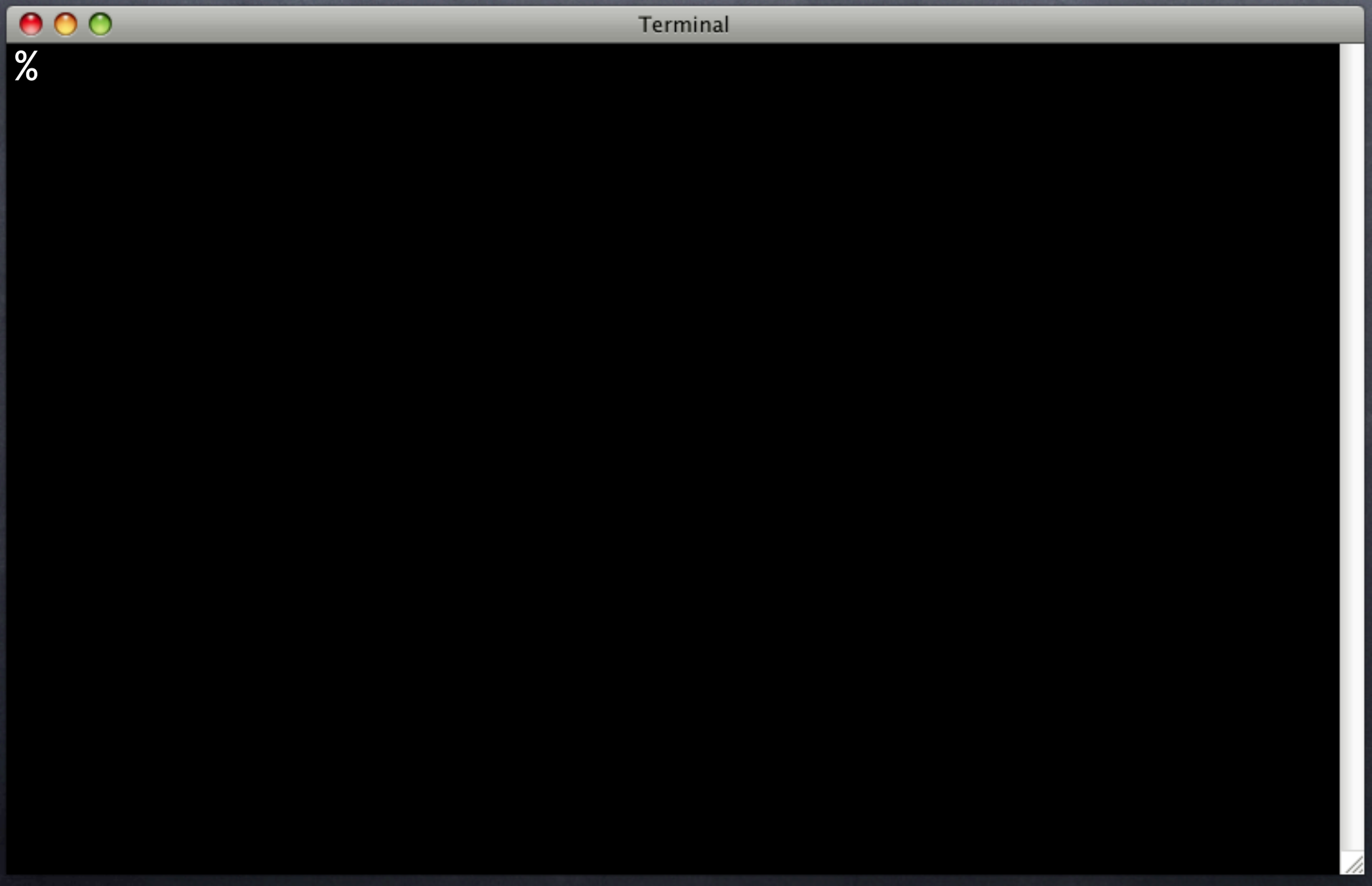


# Testing Results

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() should return active users
ok 2 - active_users() should return active users
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
ok
All tests successful.
```

# Differing Rows Diagnostics



# Differing Rows Diagnostics

Terminal

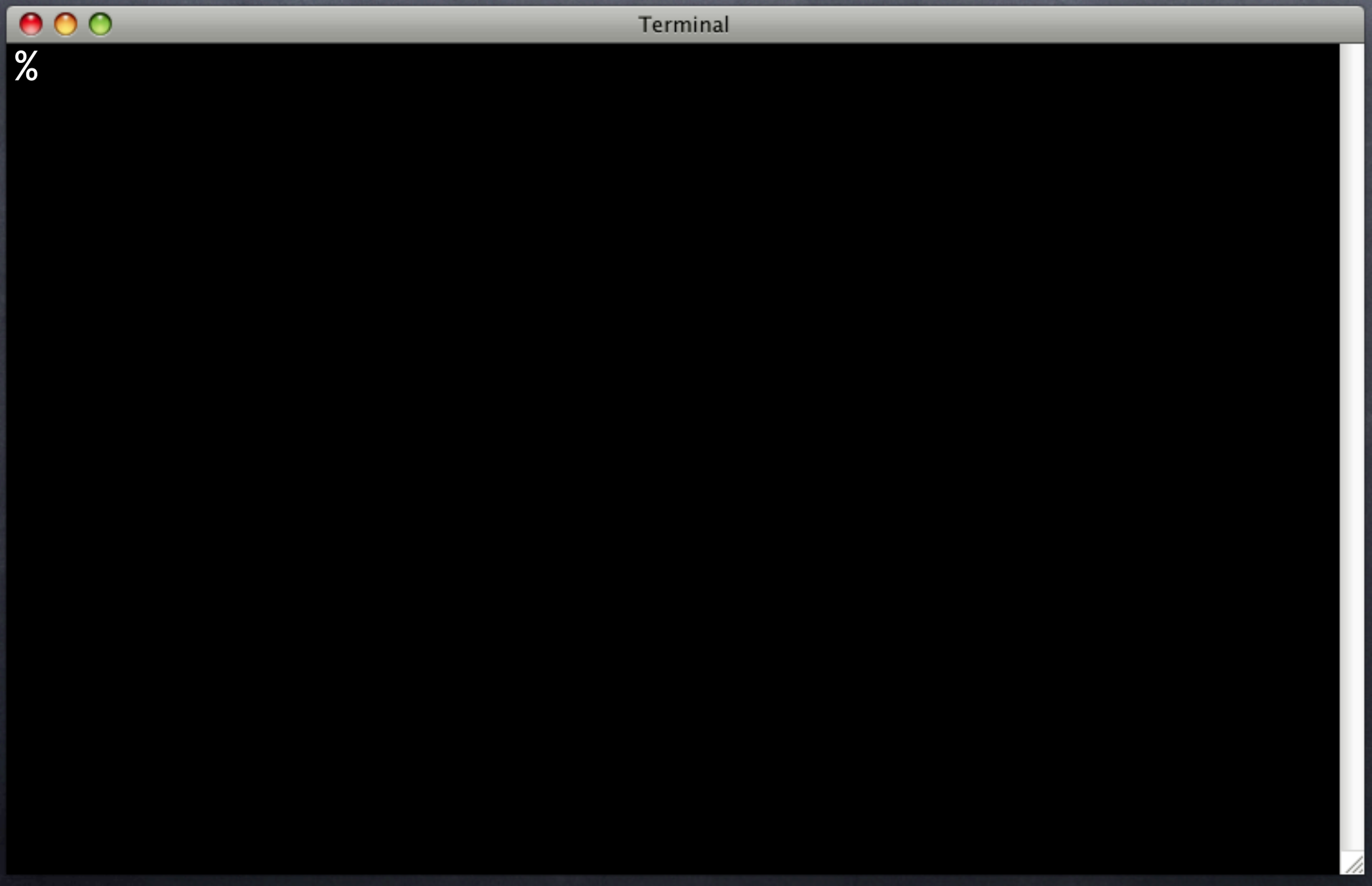
```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() should return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Results differ beginning at row 2:
#         have: (strongrrl,portland,"Julie Wheeler",t)
#         want: (strongrrl,design,"Julie Wheeler",t)
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Differing Rows Diagnostics

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() should return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Results differ beginning at row 2:
#         have: (strongrrl,portland,"Julie Wheeler",t)
#         want: (strongrrl,design,"Julie Wheeler",t)
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Differing Columns Diagnostics



# Differing Columns Diagnostics

Terminal

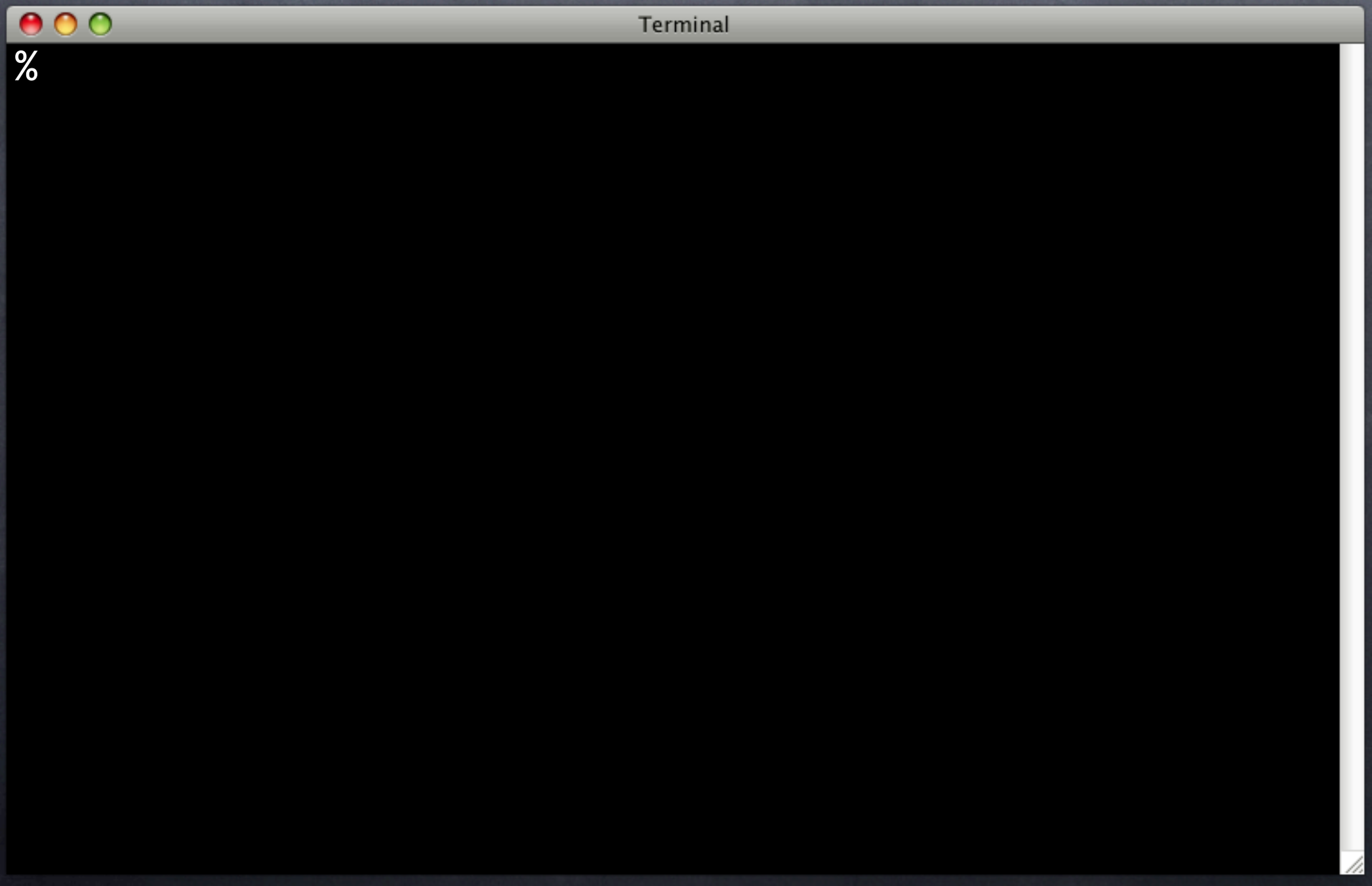
```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (anna,yddad,"Anna Wheeler",t)
#         want: (anna,yddad,"Anna Wheeler")
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Differing Columns Diagnostics

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (anna,yddad,"Anna Wheeler",t)
#         want: (anna,yddad,"Anna Wheeler")
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Missing Row Diagnostics



# Missing Row Diagnostics

Terminal

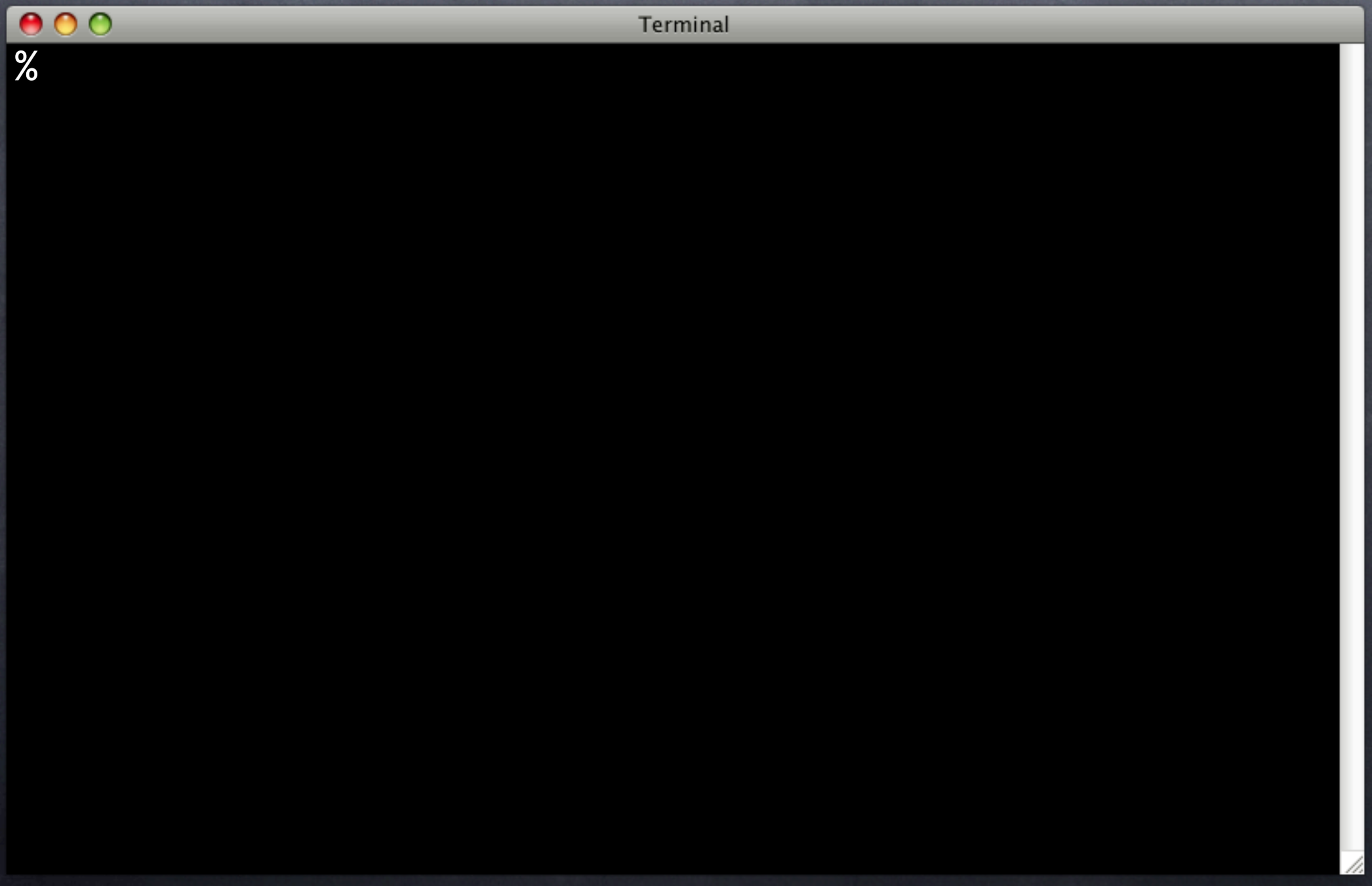
```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Results differ beginning at row 3:
#         have: (theory,s3kr1t,"David Wheeler",t)
#         want: NULL
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Missing Row Diagnostics

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Results differ beginning at row 3:
#         have: (theory,s3kr1t,"David Wheeler",t)
#         want: NULL
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Diagnostics for Differing Data Types



# Diagnostics for Differing Data Types

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (anna,yddad,"Anna Wheeler",t)
#         want: (anna,yddad,"Anna Wheeler",t)
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Diagnostics for Differing Data Types

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (anna,yddad,"Anna Wheeler",t)
#         want: (anna,yddad,"Anna Wheeler",t)
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

# Diagnostics for Differing Data Types

Terminal

```
% pg_prove -v -d try results_eq.sql
results_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (anna,yddad,"Anna Wheeler",t)
#         want: (anna,yddad,"Anna Wheeler",t)
ok 3 - We should have users
ok 4 - We should have active users
ok 5 - We should have inactive users
ok 6 - Should have active nicks
1..6
# Looks like you failed 1 test of 6
Failed 1/6 subtests
```

Say what?

# Testing Sets

# Testing Sets

- When order does not matter

# Testing Sets

- When order does not matter
- When duplicate tuples do not matter

# Testing Sets

- When order does not matter
- When duplicate tuples do not matter
- Use queries

# Testing Sets

- When order does not matter
- When duplicate tuples do not matter
- Use queries
- Or prepared statement names

# Testing Bags

# Testing Bags

- When order does not matter

# Testing Bags

- When order does not matter
- Duplicates matter

# Testing Bags

- When order does not matter
- Duplicates matter
- Use queries

# Testing Bags

- When order does not matter
- Duplicates matter
- Use queries
- Or prepared statement names


Emacs
--:--  try.sql        All  (SQL[ansi])

Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

--:--  try.sql  All  (SQL[ansi])----------

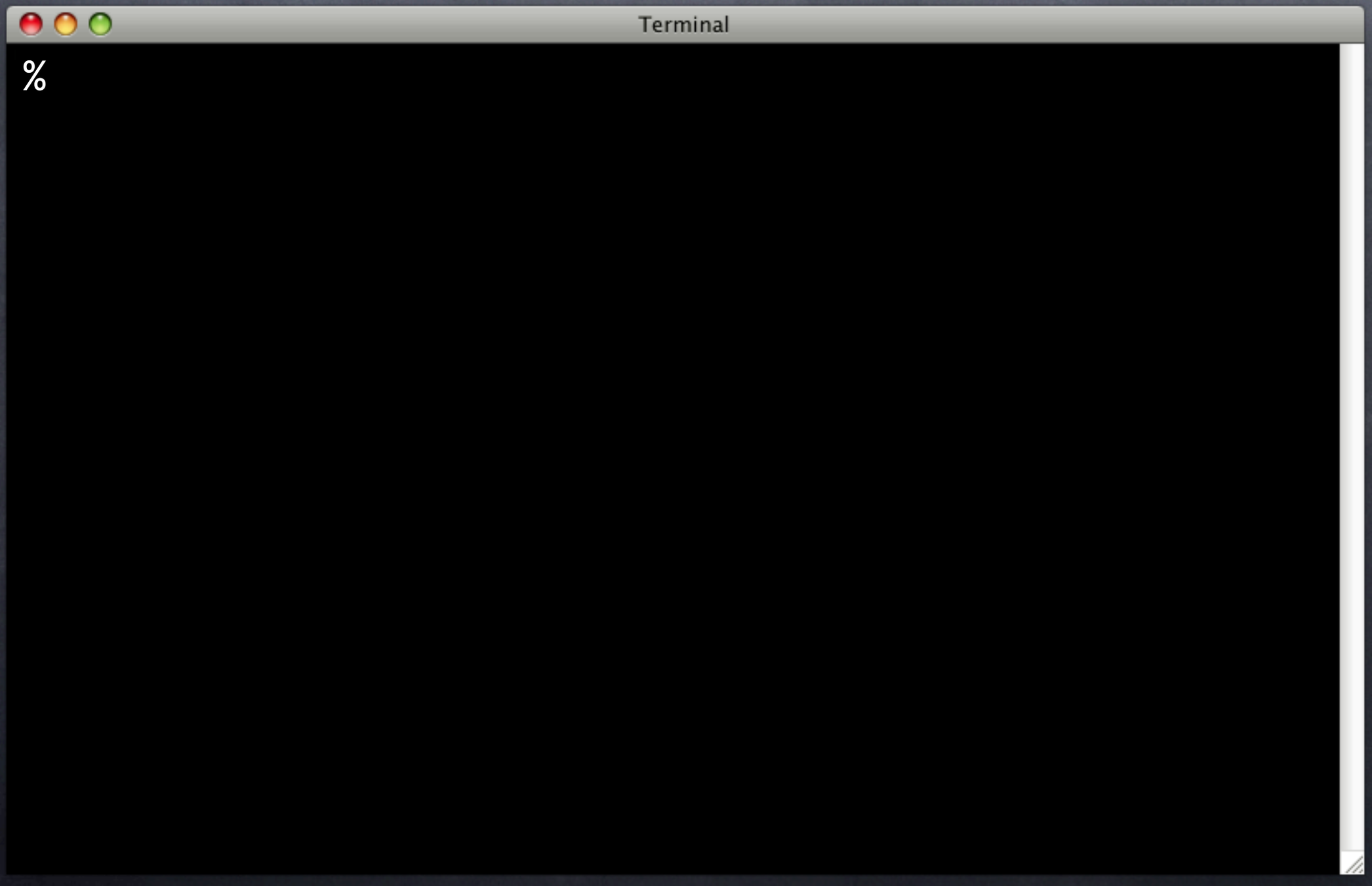
Emacs

```
SELECT set_eq(
    'SELECT * FROM active_users()',
    'SELECT * FROM users WHERE active',
    'active_users() should return active users'
);

SELECT set_eq(
    'SELECT * FROM active_users()',
    $$VALUES ('anna', 'yddad', 'Anna Wheeler', true),
             ('strongrrl', 'design', 'Julie Wheeler', true),
             ('theory', 's3kr1t', 'David Wheeler', true)
    $$,
    'active_users() should return active users'
);

PREPARE users_test AS SELECT * FROM active_users();
PREPARE users_expect AS
 SELECT * FROM users WHERE active;

SELECT bag_eq(
    'users_test',
    'users_expect',
    'We should have users'
);
```

No ORDER BY

--:--  try.sql  All  (SQL[ansi])----------

# Differing Rows Diagnostics



# Differing Rows Diagnostics

Terminal

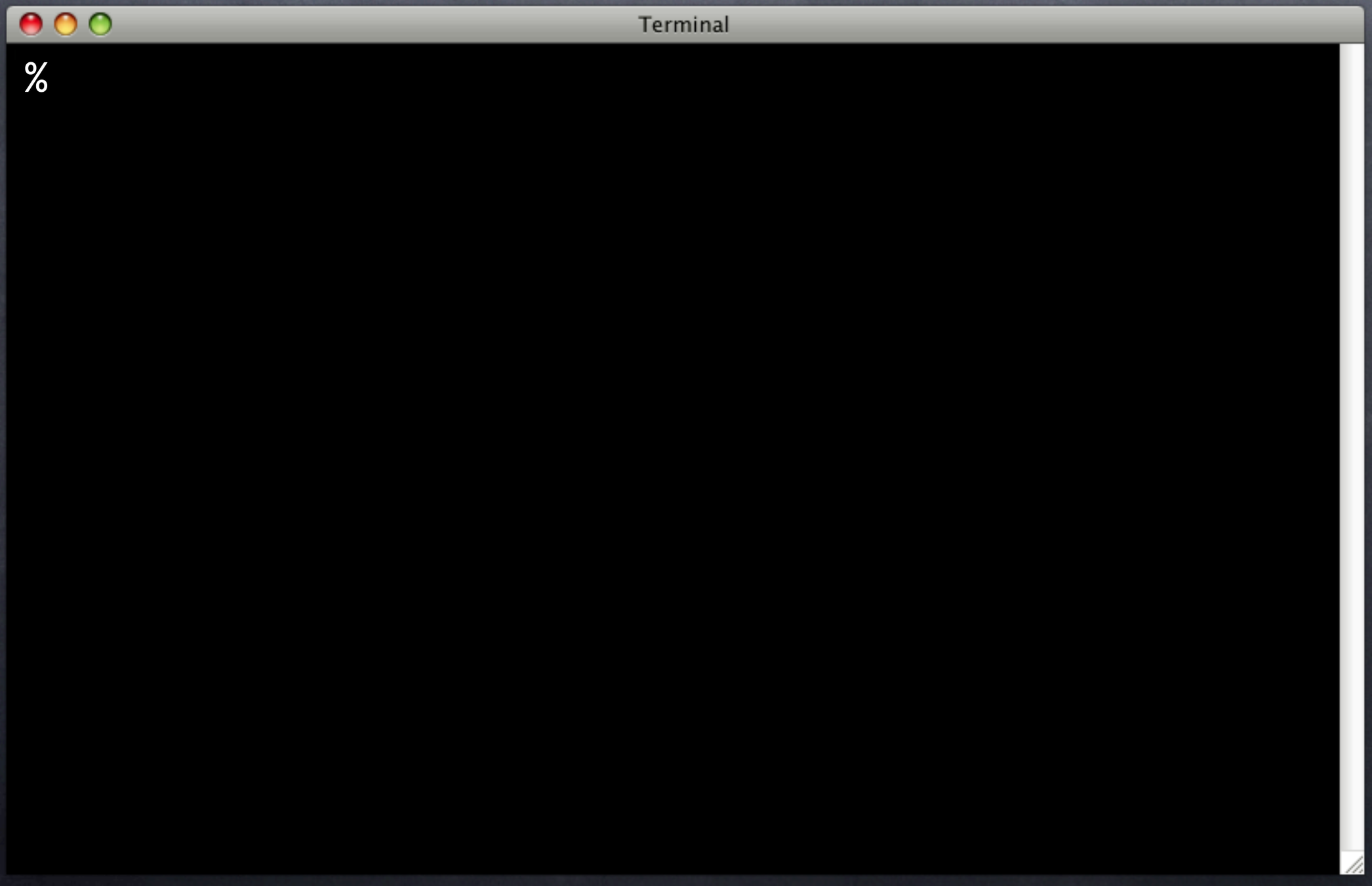
```
% pg_prove -v -d try set_eq.sql
set_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2 "active_users() return active users"
#     Extra records:
#         (87,Jackson)
#         (1,Jacob)
#     Missing records:
#         (44,Anna)
#         (86,Angelina)
ok 3 - We should have users
1..3
# Looks like you failed 1 test of 3
Failed 1/3 subtests
```

# Differing Rows Diagnostics

Terminal

```
% pg_prove -v -d try set_eq.sql
set_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2 "active_users() return active users"
#     Extra records:
#         (87,Jackson)
#         (1,Jacob)
#     Missing records:
#         (44,Anna)
#         (86,Angelina)
ok 3 - We should have users
1..3
# Looks like you failed 1 test of 3
Failed 1/3 subtests
```

# Diagnostics for Differing Data Types



# Diagnostics for Differing Data Types

Terminal

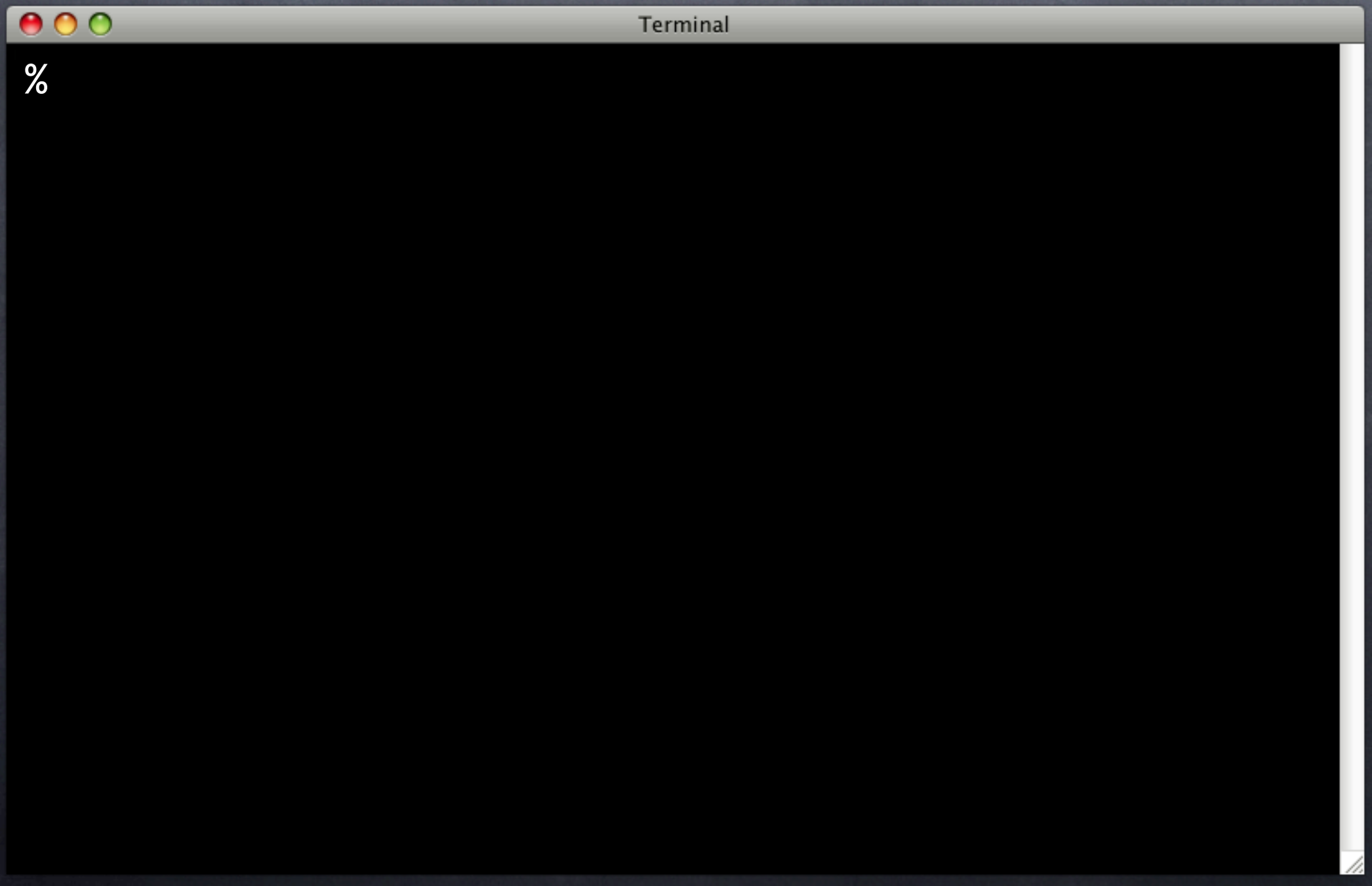
```
% pg_prove -v -d try set_eq.sql
set_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (integer,text)
#         want: (text,integer)
ok 3 - We should have users
1..3
# Looks like you failed 1 test of 3
Failed 1/3 subtests
```

# Diagnostics for Differing Data Types

Terminal

```
% pg_prove -v -d try set_eq.sql
set_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (integer,text)
#         want: (text,integer)
ok 3 - We should have users
1..3
# Looks like you failed 1 test of 3
Failed 1/3 subtests
```

# Diagnostics for Different Numbers of Columns



# Diagnostics for Different Numbers of Columns

Terminal

```
% pg_prove -v -d try set_eq.sql
set_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (integer)
#         want: (text,integer)
ok 3 - We should have users
1..3
# Looks like you failed 1 test of 3
Failed 1/3 subtests
```

# Diagnostics for Different Numbers of Columns

Terminal

```
% pg_prove -v -d try set_eq.sql
set_eq.sql ..
ok 1 - active_users() return active users
not ok 2 - active_users() return active users
# Failed test 2: "active_users() return active users"
#     Columns differ between queries:
#         have: (integer)
#         want: (text,integer)
ok 3 - We should have users
1..3
# Looks like you failed 1 test of 3
Failed 1/3 subtests
```

# Structural Testing

# Structural Testing

- Validate presence/absence of objects

# Structural Testing

- Validate presence/absence of objects
- *_are()

# Structural Testing

- Validate presence/absence of objects
- *_are()
- Specify complete lists of objects

# Structural Testing

- Validate presence/absence of objects
- *_are()
- Specify complete lists of objects
- Useful failure diagnostics

Emacs

--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
SELECT tablespaces_are(
    ARRAY[ 'dbspace', 'indexspace' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SELECT tablespaces_are(
    ARRAY[ 'dbspace', 'indexspace' ]
);

SELECT schemas_are(
    ARRAY[ 'public', 'contrib', 'biz' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SELECT tablespaces_are(
    ARRAY[ 'dbspace', 'indexspace' ]
);

SELECT schemas_are(
    ARRAY[ 'public', 'contrib', 'biz' ]
);

SELECT tables_are(
    'biz',
    ARRAY[ 'users', 'widgets' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT tablespaces_are(
    ARRAY[ 'dbspace', 'indexspace' ]
);

SELECT schemas_are(
    ARRAY[ 'public', 'contrib', 'biz' ]
);

SELECT tables_are(
    'biz',
    ARRAY[ 'users', 'widgets' ]
);

SELECT views_are(
    'biz',
    ARRAY[ 'user_list', 'widget_list' ]
);
```

```
--:--  try.sql       All  (SQL[ansi])----------
```

Emacs

```
SELECT tablespaces_are(
    ARRAY[ 'dbspace', 'indexspace' ]
);

SELECT schemas_are(
    ARRAY[ 'public', 'contrib', 'biz' ]
);

SELECT tables_are(
    'biz',
    ARRAY[ 'users', 'widgets' ]
);

SELECT views_are(
    'biz',
    ARRAY[ 'user_list', 'widget_list' ]
);

SELECT sequences_are(
    'biz',
    ARRAY[ 'users_id_seq', 'widgets_id_seq' ]
);
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
SELECT indexes_are(
    'biz', 'users',
    ARRAY[ 'users_pkey' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SELECT indexes_are(
    'biz', 'users',
    ARRAY[ 'users_pkey' ]
);

SELECT functions_are(
    'biz',
    ARRAY[ 'get_users', 'get_widgets' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------------------------------------------------------------------
```

Emacs

```
SELECT indexes_are(
    'biz', 'users',
    ARRAY[ 'users_pkey' ]
);

SELECT functions_are(
    'biz',
    ARRAY[ 'get_users', 'get_widgets' ]
);

SELECT users_are(
    ARRAY[ 'postgres', 'david', 'bric' ]
);
```

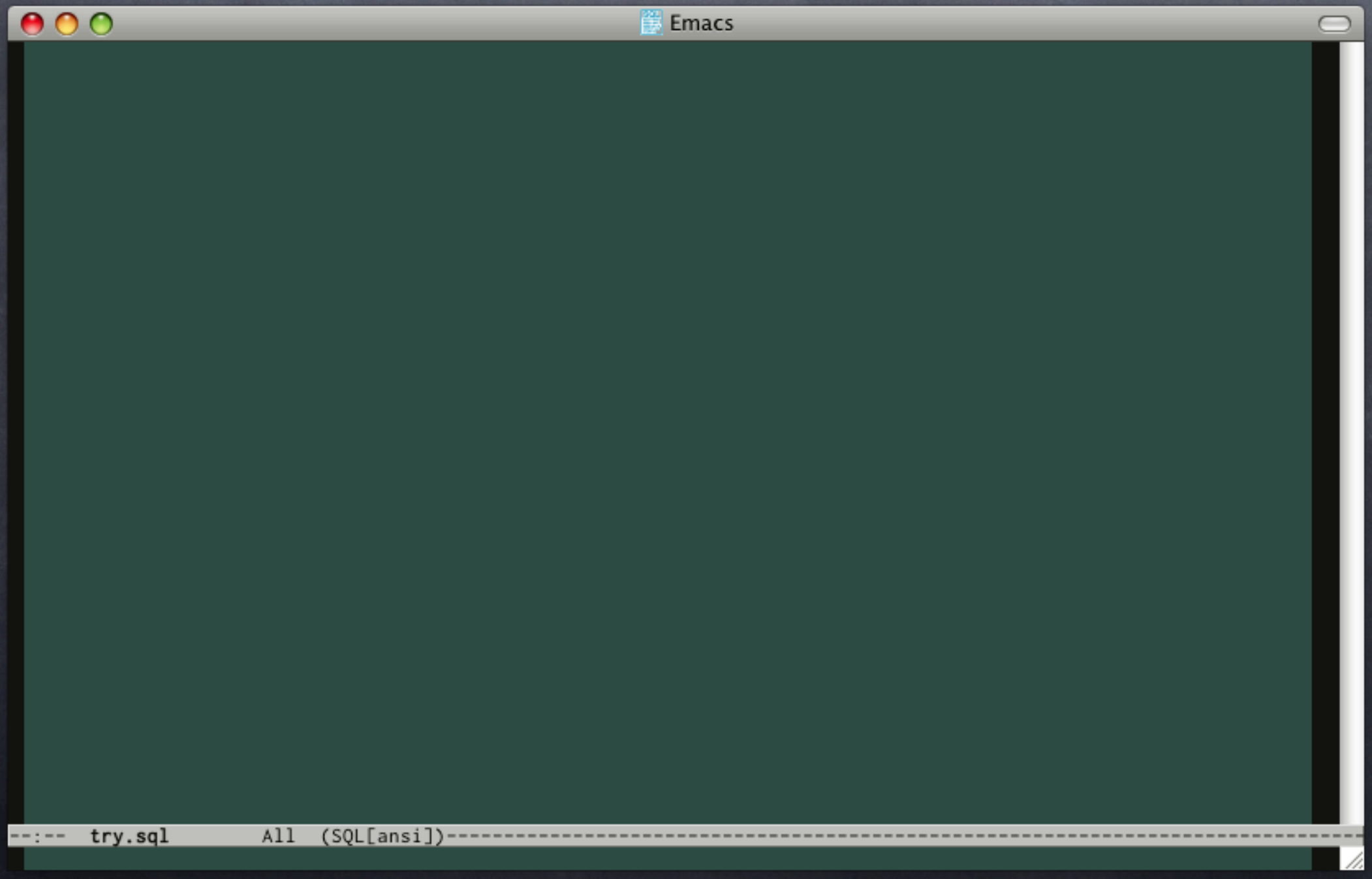
```
--:--  try.sql        All  (SQL[ansi])----------
```

Emacs

```
SELECT indexes_are(
    'biz', 'users',
    ARRAY[ 'users_pkey' ]
);

SELECT functions_are(
    'biz',
    ARRAY[ 'get_users', 'get_widgets' ]
);

SELECT users_are(
    ARRAY[ 'postgres', 'david', 'bric' ]
);

SELECT groups_are(
    ARRAY[ admins', 'devs' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Structural Testing



# Structural Testing

Emacs

```
SELECT languages_are(
    ARRAY[ 'plpgsql', 'plperlu', 'plperl' ]
);
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Structural Testing

Emacs

```
SELECT languages_are(
    ARRAY[ 'plpgsql', 'plperlu', 'plperl' ]
);

SELECT opclasses_are(
    'biz',
    ARRAY[ 'biz_ops' ]
);
```

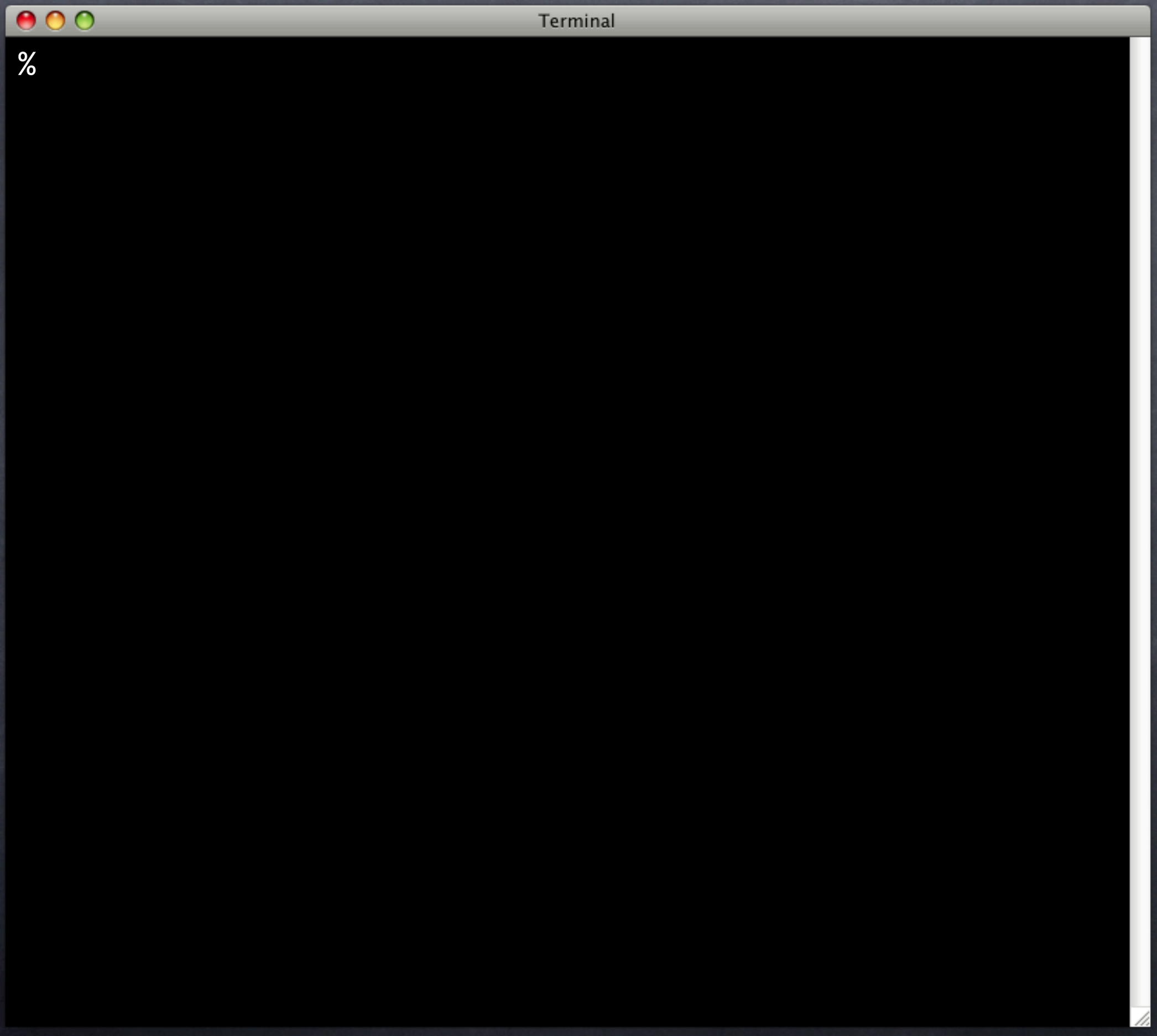
--:--  try.sql  All  (SQL[ansi])----------

# Structural Testing

Emacs

```
SELECT languages_are(
    ARRAY[ 'plpgsql', 'plperlu', 'plperl' ]
);

SELECT opclasses_are(
    'biz',
    ARRAY[ 'biz_ops' ]
);

SELECT rules_are(
    'biz', 'users',
    ARRAY[ 'ins_me', 'upd_me' ]
);
```

--:--  try.sql  All  (SQL[ansi])----------


Terminal
%

Terminal

```
% pg_prove -v -d try schema.sql
schema.sql ..
ok 1 - There should be the correct schemas
not ok 2 - Schema biz should have the correct tables
# Failed test 2: "Schema biz should have the correct
tables"
#     Extra tables:
#         mallots
#         __test_table
#     Missing tables:
#         users
#         widgets
ok 3 - Schema biz should have the correct views
ok 4 - Schema biz should have the correct sequences
ok 5 - Table biz.users should have the correct indexes
ok 6 - Schema biz should have the correct functions
ok 7 - There should be the correct users
ok 8 - There should be the correct groups
ok 9 - There should be the correct procedural languages
ok 10 - Schema biz should have the correct operator classes
ok 11 - Relation biz.users should have the correct rules
1..11
# Looks like you failed 1 test of 11
Failed 1/11 subtests
```

Terminal

```
% pg_prove -v -d try schema.sql
schema.sql ..
ok 1 - There should be the correct schemas
not ok 2 - Schema biz should have the correct tables
# Failed test 2: "Schema biz should have the correct
tables"
#     Extra tables:
#         mallots
#         __test_table
#     Missing tables:
#         users
#         widgets
ok 3 - Schema biz should have the correct views
ok 4 - Schema biz should have the correct sequences
ok 5 - Table biz.users should have the correct indexes
ok 6 - Schema biz should have the correct functions
ok 7 - There should be the correct users
ok 8 - There should be the correct groups
ok 9 - There should be the correct procedural languages
ok 10 - Schema biz should have the correct operator classes
ok 11 - Relation biz.users should have the correct rules
1..11
# Looks like you failed 1 test of 11
Failed 1/11 subtests
```

# Schema Testing

# Schema Testing

- Validate presence of objects

# Schema Testing

- Validate presence of objects
- Validate object interfaces

# Schema Testing

- Validate presence of objects
- Validate object interfaces
- Prevent others from changing schema

# Schema Testing

- Validate presence of objects
- Validate object interfaces
- Prevent others from changing schema
- Maintain a consistent interface

Emacs

--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(    'users', 'user_id' );
SELECT col_type_is(   'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(     'users', 'user_id' );

SELECT has_column(    'users', 'birthdate' );
SELECT col_type_is(   'users', 'birthdate', 'date' );
SELECT col_is_null(   'users', 'birthdate' );

SELECT has_column(      'users', 'state' );
SELECT col_type_is(     'users', 'state', 'text' );
SELECT col_not_null(    'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

```
BEGIN;
SET search_path TO public, tap;
SELECT plan(13);

SELECT has_table( 'users' );
SELECT has_pk(    'users' );

SELECT has_column(   'users', 'user_id' );
SELECT col_type_is(  'users', 'user_id', 'integer' );
SELECT col_not_null( 'users', 'user_id' );
SELECT col_is_pk(    'users', 'user_id' );

SELECT has_column(   'users', 'birthdate' );
SELECT col_type_is(  'users', 'birthdate', 'date' );
SELECT col_is_null(  'users', 'birthdate' );

SELECT has_column(     'users', 'state' );
SELECT col_type_is(    'users', 'state', 'text' );
SELECT col_not_null(   'users', 'state' );
SELECT col_default_is( 'users', 'state', 'active' );

SELECT * FROM finish();
ROLLBACK;
```

--:--  try.sql  All  (SQL[ansi])----------

Emacs

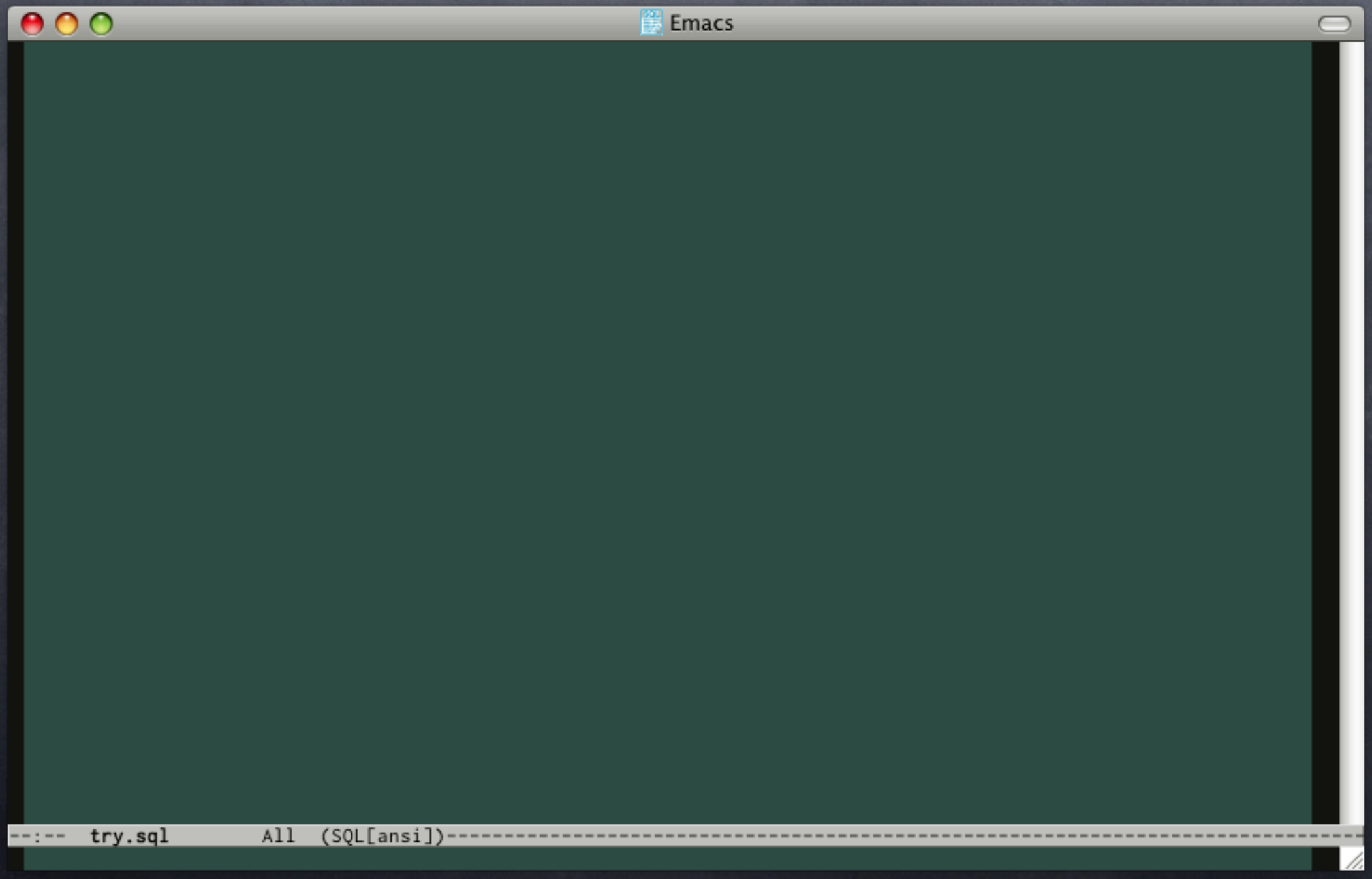
--:--  try.sql        All  (SQL[ansi])----------

Emacs

```
SELECT has_tablespace( 'indexspace',  '/data/idxs' );
SELECT has_schema( 'biz' );
SELECT has_view( 'biz', 'list_users' );
SELECT has_sequence( 'biz', 'users_id_seq' );
SELECT has_type( 'biz', 'timezone' );
SELECT has_domain( 'biz', 'timezone' );
SELECT has_enum( 'biz', 'escalation' );
SELECT has_index( 'biz', 'users', 'idx_nick' );
SELECT has_trigger( 'biz', 'users', 'md5_pass' );
SELECT has_rule( 'biz', 'list_users', 'user_ins' );
SELECT has_function( 'biz', 'get_user_data' );
SELECT has_role( 'postgres' );
SELECT has_user( 'postgres' );
SELECT has_group( 'postgres' );
SELECT has_language( 'plpgsql' );

SELECT fk_ok(
    'biz', 'widgets', 'user_id',
    'biz', 'users',   'id'
);
```

```
--:--  try.sql        All  (SQL[ansi])----------
```

# Function Testing Functions



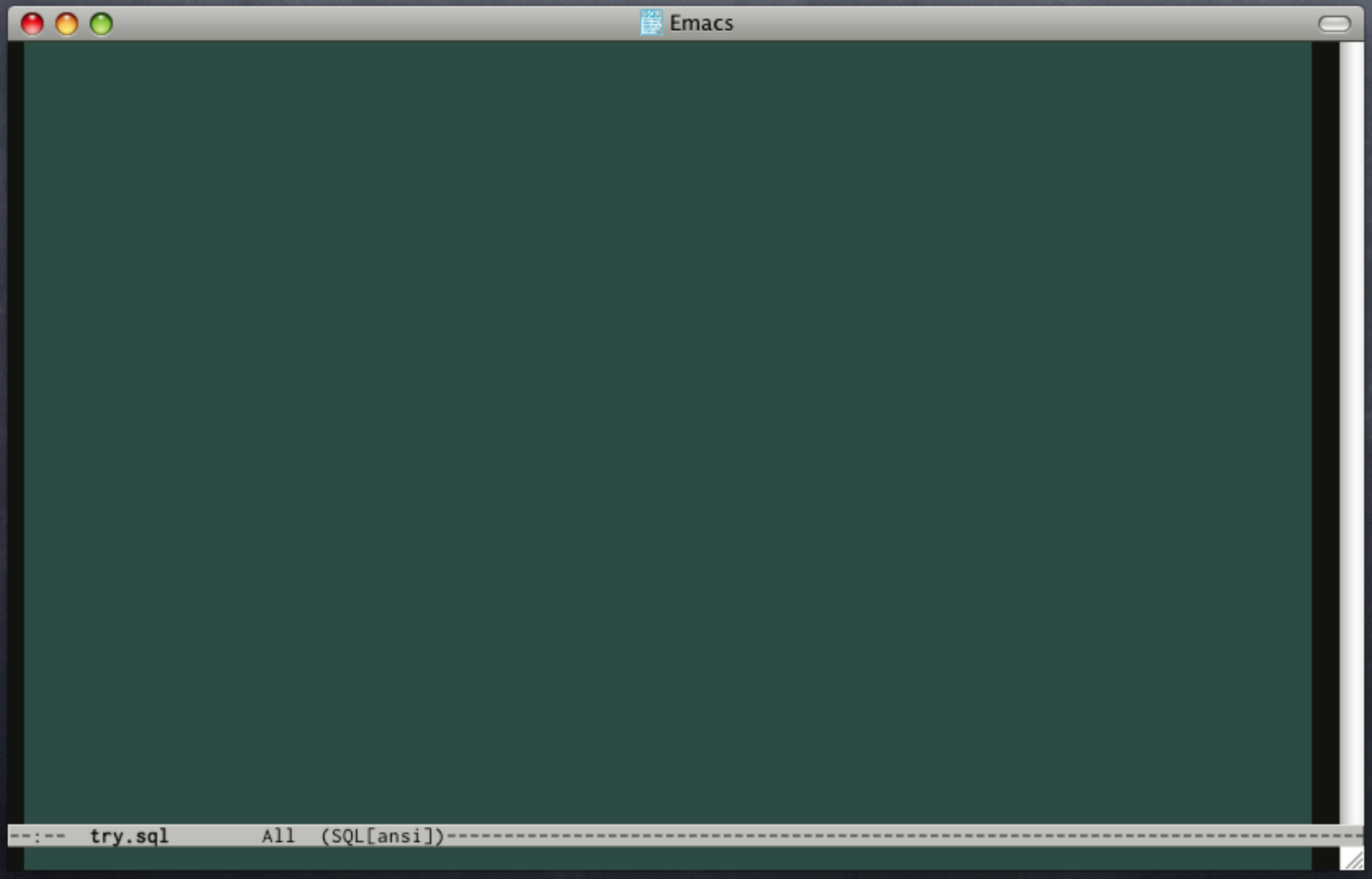
# Function Testing Functions

Emacs

```
SELECT function_lang_is( 'biz', 'get_user', 'sql' );
SELECT function_returns( 'biz', 'get_user', 'users' );
SELECT is_definer( 'biz', 'get_user_data' );
SELECT is_strict( 'biz', 'get_user_data' );
SELECT volatility_is(  'biz', 'get_user', 'volatile' );
SELECT is_aggregate( 'biz', 'array_accum' );

SELECT trigger_is(
    'biz', 'users', 'md5_pass',
    'biz', 'md5er'
);
```

--:--  try.sql  All  (SQL[ansi])----------

# Utilities



# Utilities

Emacs

```
SELECT CASE WHEN pgtap_version() < 0.17
    THEN skip('No sequence assertions before pgTAP 0.17')
    ELSE has_sequence('my_big_seq')
    END;

SELECT CASE WHEN pg_version_num() < 80100
    THEN skip('throws_ok() not supported before 8.1')
    ELSE throws_ok( 'SELECT 1/0', 22012, 'div by zero' )
    END;
```

--:--  try.sql  All  (SQL[ansi])----------

# Utilities

Emacs

```
SELECT CASE WHEN pgtap_version() < 0.17
    THEN skip('No sequence assertions before pgTAP 0.17')
    ELSE has_sequence('my_big_seq')
    END;

SELECT CASE WHEN pg_version_num() < 80100
    THEN skip('throws_ok() not supported before 8.1')
    ELSE throws_ok( 'SELECT 1/0', 22012, 'div by zero' )
    END;
```

--:--  try.sql  All  (SQL[ansi])----------

# Utilities

Emacs

```
SELECT CASE WHEN pgtap_version() < 0.17
    THEN skip('No sequence assertions before pgTAP 0.17')
    ELSE has_sequence('my_big_seq')
    END;

SELECT CASE WHEN pg_version_num() < 80100
    THEN skip('throws_ok() not supported before 8.1')
    ELSE throws_ok( 'SELECT 1/0', 22012, 'div by zero' )
    END;
```

--:--  try.sql  All  (SQL[ansi])----------

# Other Features and Topics

# Other Features and Topics

- xUnit-Style testing

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development
- Integration with Perl unit tests

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development
- Integration with Perl unit tests
- Integration with pg_regress

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development
- Integration with Perl unit tests
- Integration with pg_regress
- Negative assertions

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development
- Integration with Perl unit tests
- Integration with pg_regress
- Negative assertions
- Constraint assertions

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development
- Integration with Perl unit tests
- Integration with pg_regress
- Negative assertions
- Constraint assertions
- Roles and permissions assertions

# Other Features and Topics

- xUnit-Style testing
- Test-Driven development
- Integration with Perl unit tests
- Integration with pg_regress
- Negative assertions
- Constraint assertions
- Roles and permissions assertions
- Lots more!

# Thank You

## pgTAP Best Practices

David E. Wheeler

Kineticode, Inc.
PostgreSQL Experts, Inc.

PostgreSQL Conference West 2009