Testing

`https://www.sqlite.org/testing.html`

> As of version 3.33.0 (2020-08-14), the SQLite library consists of approximately 143.4 KSLOC of C code. [...] By comparison, the project has 640 times as much test code and test scripts [...]

Testing

https://www.sqlite.org/testing.html

- Four independently developed test harnesses
- 100% branch test coverage in an as-deployed configuration
- Millions and millions of test cases
- Out-of-memory tests
- I/O error tests
- Crash and power loss tests
- Fuzz tests
- Boundary value tests
- Disabled optimization tests
- Regression tests
- Malformed database tests
- Extensive use of assert() and run-time checks
- Valgrind analysis
- Undefined behavior checks
- Checklists

Testing

`https://www.sqlite.org/testing.html`

- Four independently developed test harnesses
- 100% branch test coverage in an as-deployed configuration
- Millions and millions of test cases
- Out-of-memory tests
- I/O error tests
- Crash and power loss tests
- Fuzz tests
- Boundary value tests
- Disabled optimization tests
- Regression tests
- Malformed database tests
- Extensive use of assert() and run-time checks
- Valgrind analysis
- Undefined behavior checks
- Checklists

Test Generation

Unit tests cover what you thought might go wrong

Generated tests can find what you didn't think about

Test Generation

Test generation needs:

- a way to generate inputs
- a driver to send the inputs and receive outputs
- a way to decide whether the output was good

Test Generation

Test generation needs:

- a way to generate inputs
- a driver to send the inputs and receive outputs
- a way to decide whether the output was good

oracle

Test Generation

Test generation needs:

- a way to generate inputs
- a driver to send the inputs and receive outputs
- a way to decide whether the output was good

Using another implementation:
differential testing

Fuzzing

Fuzzing refers to

- sending randomly generated inputs
    - typically text
- minimal result validation
    - typically: did the program crash?

Fuzzing

Fuzzing refers to

- sending randomly gene
  - typically text
- minimal result validation
  - typically: did the program crash?

```
// Makes a string of up to 31 random bytes
static std::string random_bytes() {
  std::string word = "";
  for (int i = rand() % 32; i-- > 0; )
    word += rand() % 256;
  return word;
}
```

Fuzzing

Fuzzing refers to

- sending randomly gene
  - typically text
- minimal result validation
  - typically: did the program crash?

```
// Makes a string of up to 31 random bytes
static std::string random_bytes() {
  std::string word = "";
  for (int i = rand() % 32; i-- > 0; )
    word += rand() % 256;
  ret
}
```

returns a "random" `int`

Fuzzing

Fuzzing refers to

- sending randomly ge
  - typically text
- minimal result validation
  - typically: did the program crash?

```
// Makes a string of up to 31 random bytes
static std::string random_bytes() {
  std::string word = "";
  for (int i = rand() % 32; i-- > 0; )
    word += rand() % 256;
```

use something like `srand(clock())` to generate varying values

Fuzzing

Fuzzing refers to

- sending randomly gene
  - typically text
- minimal result validation
  - typically: did the program crash?

```
// Makes a string of up to 31 random bytes
static std::string random_bytes() {
  std::string word = "";
  for (int i = rand() % 32; i-- > 0; )
    word += rand() % 256;
  return
}
```

a number from 0 to 31

Fuzzing

Fuzzing refers to

- sending randomly gene
  - typically text
- minimal result validation
  - typically: did the program crash?

```
// Makes a string of up to 31 random bytes
static std::string random_bytes() {
  std::string word = "";
  for (int i = rand() % 32; i-- > 0; )
    word += rand() % 256;
  return word;
}
```

a number from 0 to 255

Fuzzing

Fuzzing refers to

- sending randomly generated inputs
  - typically text
- minimal result validation
  - typically: did the program crash?

Fuzzing is a good idea for testing parsers, but just generating random strings is unlikely to generate many interesting MSDscript expressions

Generating Expressions

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

Possible strategy:

- randomly pick a case
- for ⟨number⟩, randomly pick one
- for ⟨variable⟩, randomly generate one
- for others, recur for nested ⟨expr⟩

Generating Expressions

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

Possible strategy:

- randomly pick a case
- for ⟨number⟩, randomly pick one
- for ⟨variable⟩, randomly generate one
- for others, recur for nested ⟨expr⟩

Likely to generate bound variables?

Generating Expressions

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

Generate **`Expr`** values or strings?

By generating strings, we can make the test generator more separate from the code it's trying to test

Simple Generator

First try — just generate numbers

```
std::string random_expr_string() {
  return std::to_string(rand());
}
```

Could check:

- **`--interp`** mode prints the same number
- **`--print`** mode prints the same number
- **`--pretty-print`** mode prints the same number
- exit code is always 0

Simple Generator

```
std::string random_expr_string() {
  if ((rand() % 10) < 6)
    return std::to_string(rand());
  else
    return random_expr_string() + "+" + random_expr_string();
}
```

Simple Generator

```
std::string random_expr_string() {
  if ((rand() % 10) < 6)
    return std::to_string(rand());
  else
    return random_expr_string() + "+" + random_expr_string();
}
```

60% of the time

Simple Generator

```
std::string random_expr_string() {
  if ((rand() % 10) < 6)
    return std::to_string(rand());
  else
    return random_expr_string() + "+" + random_expr_string();
}
```

60% of the time

40% of the time

Even without tracking the expected sum, but could check:

- **`--interp`** mode prints *some* number
- **`--print`** mode prints some expression that interps to the same number
- **`--pretty-print`** mode prints some expression that interps to the same number and pretty-prints exactly the same
- exit code is always 0

Trying Generated Expressions

Overall:

- generate an expression string
- send string as input to **`msdscript`**
- check **`msdscript`** output and exit code

Inside the **`msdscript`** implementation, we take control of input and output using **`std::istream&`** and **`std::ostream&`** arguments

From the outside, we need a way to run a program, send it input to **`std::cin`**, and capture its output to **`std::count`**

Test-Runner Helper

Provided by **`exec.cpp`**:

```
ExecResult exec_program(int argc, char **argv, std::string in);

class ExecResult {
public:
  int exit_code;
  std::string out;
  std::string err;
};
```

Test-Runner Helper

Provided by **`exec.cpp`**:

`argv[0]` is the program to run

```
ExecResult exec_program(int argc, char **argv, std::string in);

class ExecResult {
public:
  int exit_code;
  std::string out;
  std::string err;
};
```

Test-Runner Helper

Provided by **`exec.cpp`**:

```
ExecResult exec_program(int argc, char **argv, std::string in);

class ExecResult {
public:
  int exit_code;
  std::string out;
  std::string err;
};
```

one string as input

Test-Runner Helper

Provided by **`exec.cpp`**:

```
ExecResult exec_program(int argc, char **argv, std::string in);

class ExecResult {
public:
  int exit_code;
  std::string out;
  std::string err;
};
```

one string as input

two strings as output
but either might be empty

Test-Runner Helper

```
const char * const wc_argv[] = { "/usr/bin/wc", "-w" };

ExecResult wc_result = exec_program(2, wc_argv, "a b c");

if (wc_result.exit_code != 0)
  std::cerr << "non-zero exit: " << wc_result.exit_code << "\n";

if (wc_result.out != "       3\n")
  std::cerr << "bad wc result\n";
```

Another Simple Test Driver

```
int main(int argc, char **argv) {
  const char * const interp_argv[] = { "msdscript", "--interp" };
  const char * const print_argv[] = { "msdscript", "--print" };

  for (int i = 0; i < 100; i++) {
    std::string in = random_expr_string();
    std::cout << "Trying " << in << "\n";

    ExecResult interp_result = exec_program(2, interp_argv, in);
    ExecResult print_result = exec_program(2, print_argv, in);

    ExecResult interp_again_result = exec_program(2, interp_argv, print_result.out);
    if (interp_again_result.out != interp_result.out)
      throw std::runtime_error("different result for printed");
  }

  return 0;
}
```

Simple Test Driver

```
int main(int argc, char **argv) {
  const char * const interp1_argv[] = { "msdscript", "--interp" };
  const char * const interp2_argv[] = { "msdscript2", "--interp" };

  for (int i = 0; i < 100; i++) {
    std::string in = random_expr_string();
    std::cout << "Trying " << in << "\n";

    ExecResult interp1_result = exec_program(2, interp1_argv, in);
    ExecResult interp2_result = exec_program(2, interp2_argv, in);

    if (interp1_result.out != interp2_result.out)
      throw std::runtime_error("different results");
  }

  return 0;
}
```