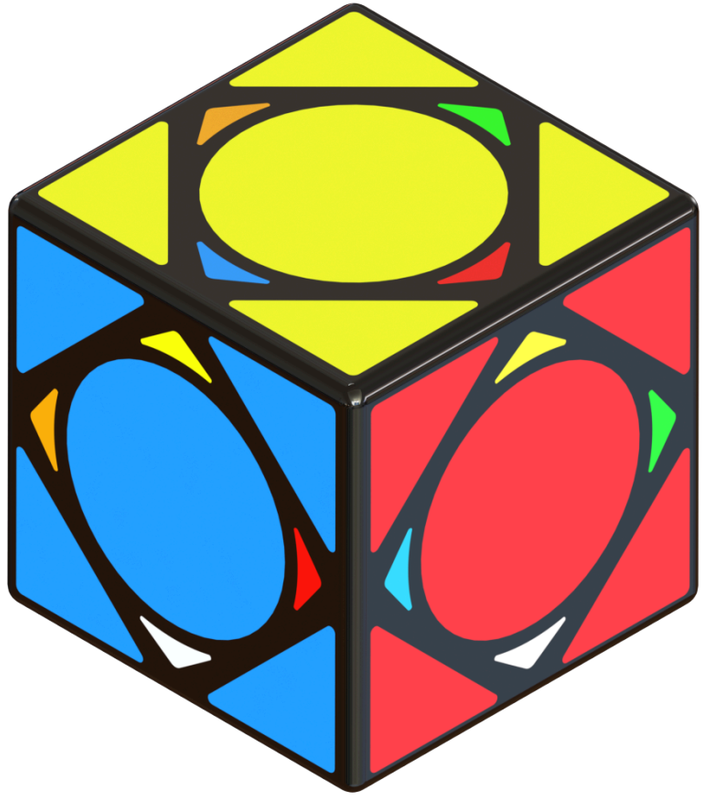
C++ now

Unit Testing an Expression Template Library in C++20

Braden Ganetsky

2024

Unit testing an expression template library in C++20

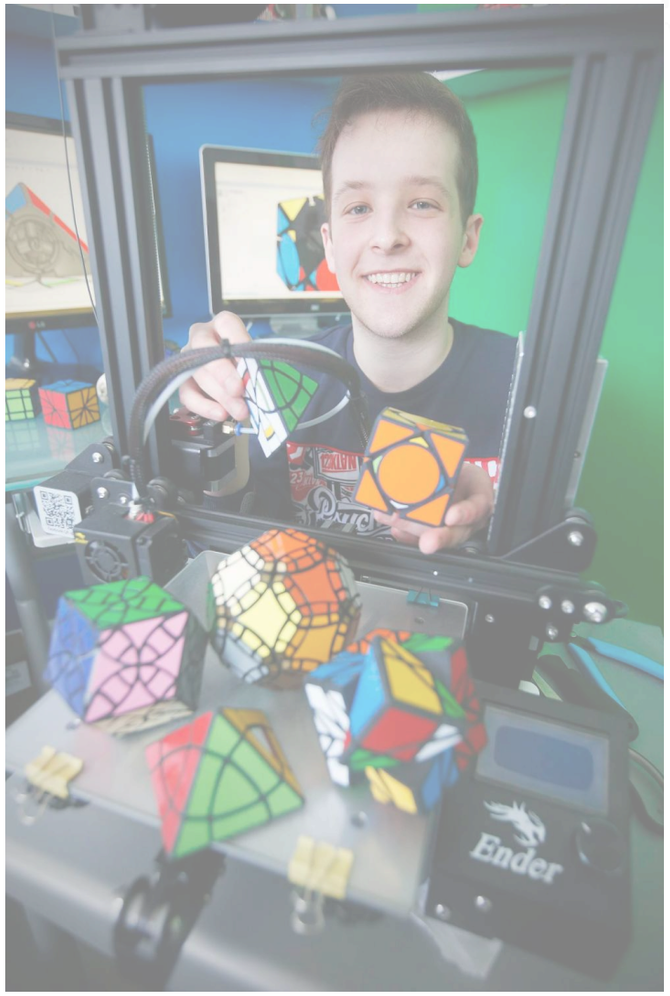


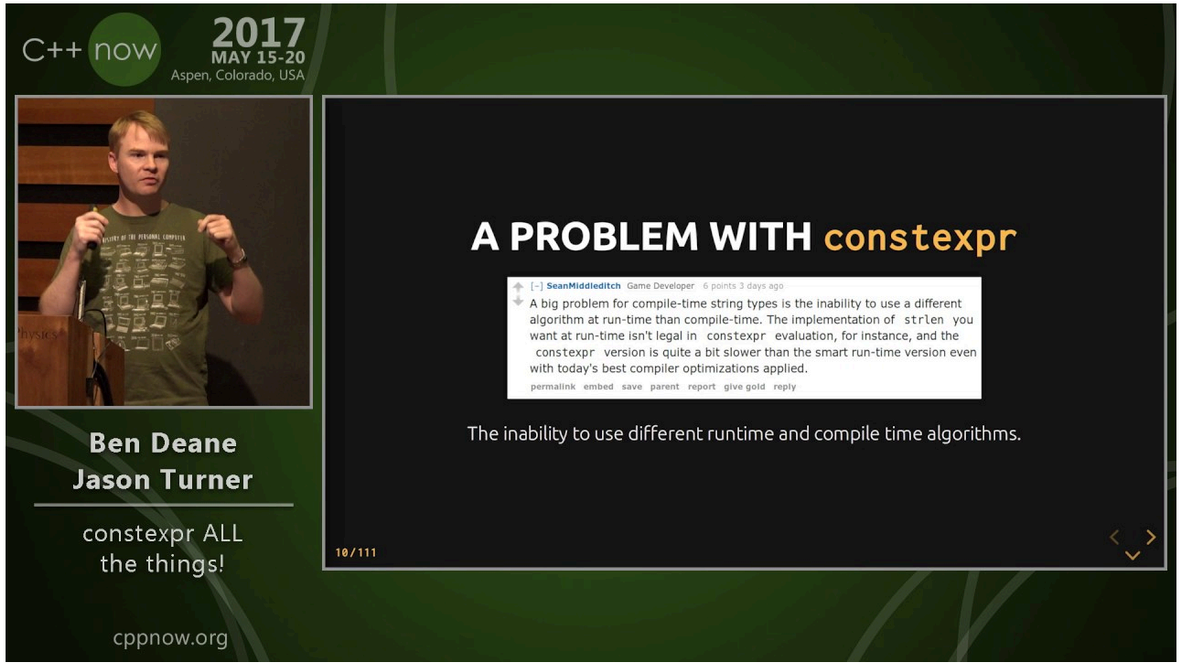
Braden Ganetsky

C++Now 2024

About me

- Canadian
- Mechanical engineering degree
- I like twisty puzzles
- Working in C++ for 3 years
- On WG21 since late 2023
- Importantly for this talk, parsing!

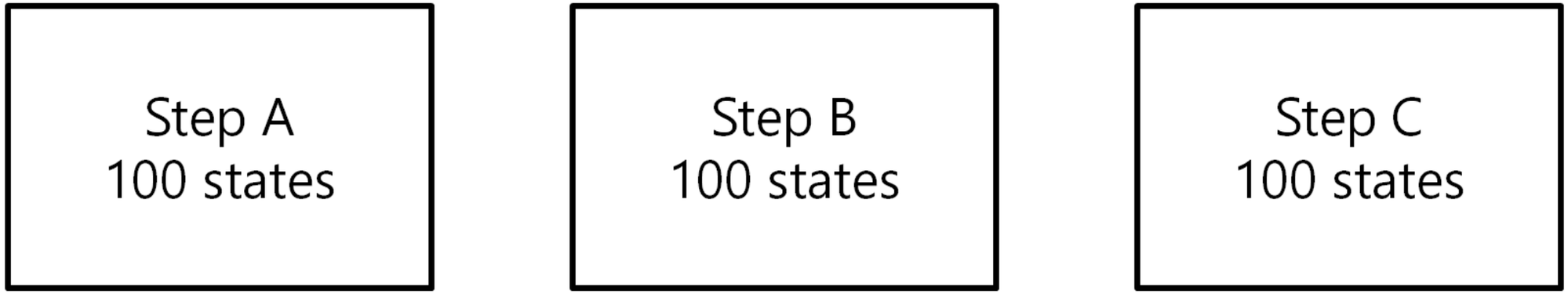




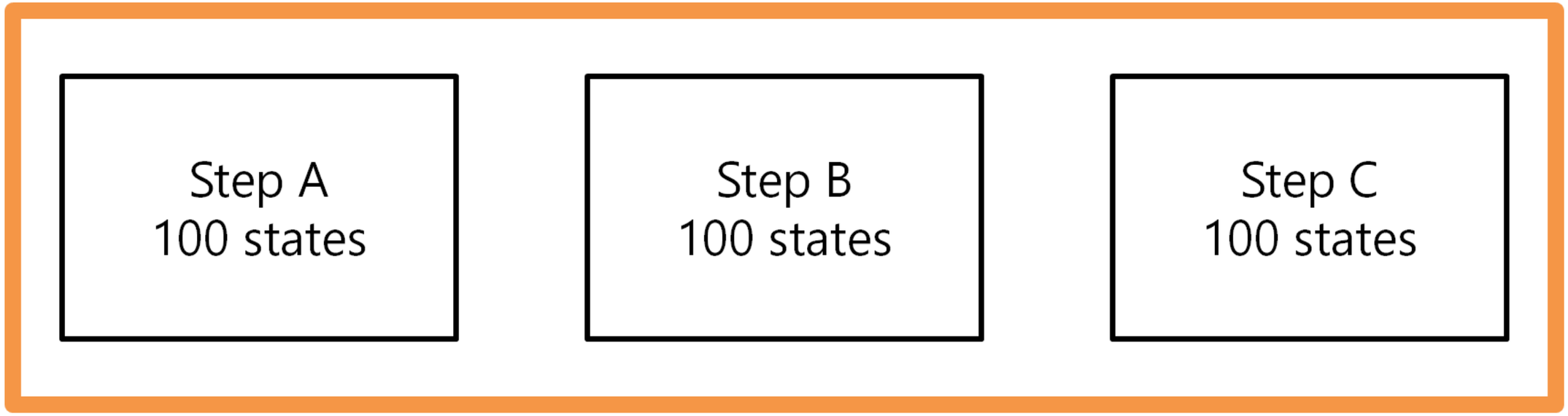
"constexpr ALL the things!"

https://www.youtube.com/watch?v=HMB9oXFobJc

Unit testing

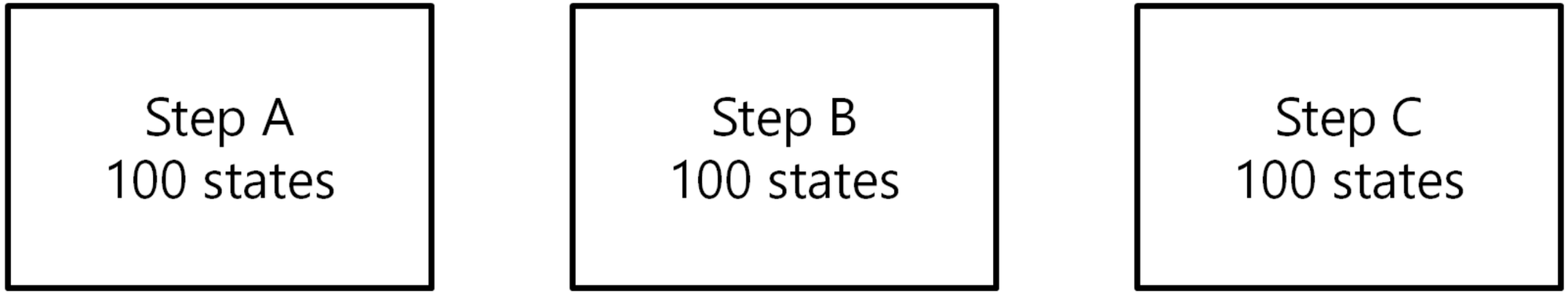


Unit testing



100 * 100 * 100 = 1,000,000 total tests

Unit testing



100 + 100 + 100 = 300 total tests

Overview

- Demo my `tok3n` library
- Test at compile-time in multiple frameworks
- Show my own framework
- Upcoming C++ features

Compile-time versus run-time

```cpp
struct Structural {
  int i;
  constexpr Structural(int i)
    : i(i) {}
};

template <auto val>
concept True = true;

template <Structural st>
requires True<st>
struct Checker {
  constexpr bool check(int i) {
    return i == st.i;
  }
};
```

```cpp
[] {
  Checker<1> chk;
  static_assert(chk.check(1));
  assert(chk.check(0));
}();
```

Compile-time versus run-time

- I claim `tok3n` is fully capable at compile-time
- Therefore it should be tested at compile-time

Goals of this talk

Need

- Test features at compile-time and run-time
- Helpful error messages at run-time

Want

- Write test only once, compile-time and run-time together
- Equally helpful error messages from compile-time cases

`tok3n` library

`tok3n` library

- Parser combinators
- Expression templates in C++20 (empty types)
- Fully capable at compile-time
- Parses span of anything, not just char

Basic usage

```
auto parser = "abc"_any_of >> "123"_all_of;
```

```
auto result = parser.parse("b1234");
```

```
assert(result.has_value());
assert(*result == std::tuple("b", "123"));
assert(result.remaining() == "4");
```

Basic usage

```
auto parser = any_of<'a','b','c'> >> all_of<'1','2','3'>;
```

```
auto result = parser.parse("b1234");
```

```
assert(result.has_value());
assert(*result == std::tuple("b", "123"));
assert(result.remaining() == "4");
```

Basic usage

```
auto parser = "abc"_any_of >> "123"_all_of;
```

```
auto result = parser.parse("b1234");
```

```
assert(result.has_value());
assert(*result == std::tuple("b", "123"));
assert(result.remaining() == "4");
```

Basic usage

```
auto parser = "abc"_any_of >> "123"_all_of;
```

```
constexpr auto result = parser.parse("b1234");
```

```
static_assert(result.has_value());
static_assert(*result == std::tuple("b", "123"));
static_assert(result.remaining() == "4");
```

Basic usage

```
auto parser = "abc"_any_of | "123"_all_of;
```

```
constexpr auto result = parser.parse("b1234");
```

```
static_assert(result.has_value());
static_assert(*result == "b");
static_assert(result.remaining() == "1234");
```

Basic usage

```
auto parser = "abc"_any_of | "123"_all_of;
```

```
constexpr auto result = parser.parse("1234b");
```

```
static_assert(result.has_value());
static_assert(*result == "123");
static_assert(result.remaining() == "4b");
```

Basic usage

```
auto parser = +"abc"_any_of;
```

```
auto result = parser.parse("abcbad");
```

```
assert(result.has_value());
assert((*result == std::vector<Output<char>>{"a", "b", "c", "b", "a"}));
assert(result.remaining() == "d");
```

Basic usage

```
auto parser = *"abc"_any_of;
```

```
auto result = parser.parse("abcbad");
```

```
assert(result.has_value());
assert((*result == std::vector<Output<char>>{"a", "b", "c", "b", "a"}));
assert(result.remaining() == "d");
```

Basic usage

```
auto parser = ~"abc"_any_of;
```

```
constexpr auto result = parser.parse("abc");
```

```
static_assert(result.has_value());
static_assert(*result == std::optional<Output<char>>("a"));
static_assert(result.remaining() == "bc");
```

Basic usage

```
auto parser = ~"abc"_any_of;
```

```
constexpr auto result = parser.parse("xyz");
```

```
static_assert(result.has_value());
static_assert(*result == std::nullopt);
static_assert(result.remaining() == "xyz");
```

Modifiers

```
auto parser = join("abc"_any_of >> "123"_all_of);
```

```
constexpr auto result = parser.parse("b1234");
```

```
static_assert(result.has_value());
static_assert(*result == "b123");
static_assert(result.remaining() == "4");
```

Modifiers

```
auto parser = ("abc"_any_of >> "123"_all_of) % join;
```

```
constexpr auto result = parser.parse("b1234");
```

```
static_assert(result.has_value());
static_assert(*result == "b123");
static_assert(result.remaining() == "4");
```

Modifiers

```
auto parser = join(exactly<3>("abc"_any_of) >> "123"_all_of);
```

```
constexpr auto result = parser.parse("cba123");
```

```
static_assert(result.has_value());
static_assert(*result == "cba123");
static_assert(result.remaining() == "");
```

Modifiers

```
auto parser = ignore("abc"_any_of) >> "123"_all_of;
```

```
constexpr auto result = parser.parse("b1234");
```

```
static_assert(result.has_value());
static_assert(*result == "123");
static_assert(result.remaining() == "4");
```

Modifiers

```
auto digit = "0123456789"_any_of;
auto dot = ignore("."_all_of);
auto parser = digit >> dot >> digit >> dot >> digit;
```

```
constexpr auto result = parser.parse("1.2.3");
```

```
static_assert(result.has_value());
static_assert(*result == std::tuple("1", "2", "3"));
static_assert(result.remaining() == "");
```

Modifiers

```
constexpr auto is_one = [](auto span) { return span.front() == '1'; };

auto parser = "01"_any_of % fn<is_one> % exactly<3>;

constexpr auto result = parser.parse("1010");

static_assert(result.has_value());
static_assert(*result == std::array{ true, false, true });
static_assert(result.remaining() == "0");
```

Non-char spans

```
std::vector<int> vec{ 2, 5, 1, 3, 1, 4, 7, 1, 9, 3 };
```

```
auto even = any_of<0,2,4,6,8>;
auto odd = any_of<1,3,5,7,9>;
auto parser = ignore(*even) >> join(+odd);
```

```
auto result = parser.parse(vec);
```

```
assert(result.has_value());
assert(*result == std::vector{ 5, 1, 3, 1 });
assert(result.remaining() == std::vector{ 4, 7, 1, 9, 3 });
```

Non-char spans

```
enum class TokenKind { ident, num, lbrace, ... };

class Token {
  TokenKind kind;
  ... // other members

  constexpr bool operator==(TokenKind other) {
    return kind == other;
  }
}
```

```
std::vector<Token> tokens = call_tokenizer(...);
```

```
auto qualified_name = all_of<TokenKind::ident>
                    % delimit(all_of<TokenKind::double_colon>);
...
auto function_declaration = qualified_name >> ...;

auto result = function_declaration.parse(tokens);
```

Recursion with JSON

Introducing JSON

العربية Български 中文 Český Dansk Nederlands English Esperanto Français Deutsch Ελληνικά עברית Magyar Indonesia Italiano 日本
한국어 فارسی Norsk Polski Português Română Русский Српско-хрватски Slovenščina Español Svenska Türkçe Українська Tiếng Việt

ECMA-404 The JSON Data Interchange Standard.

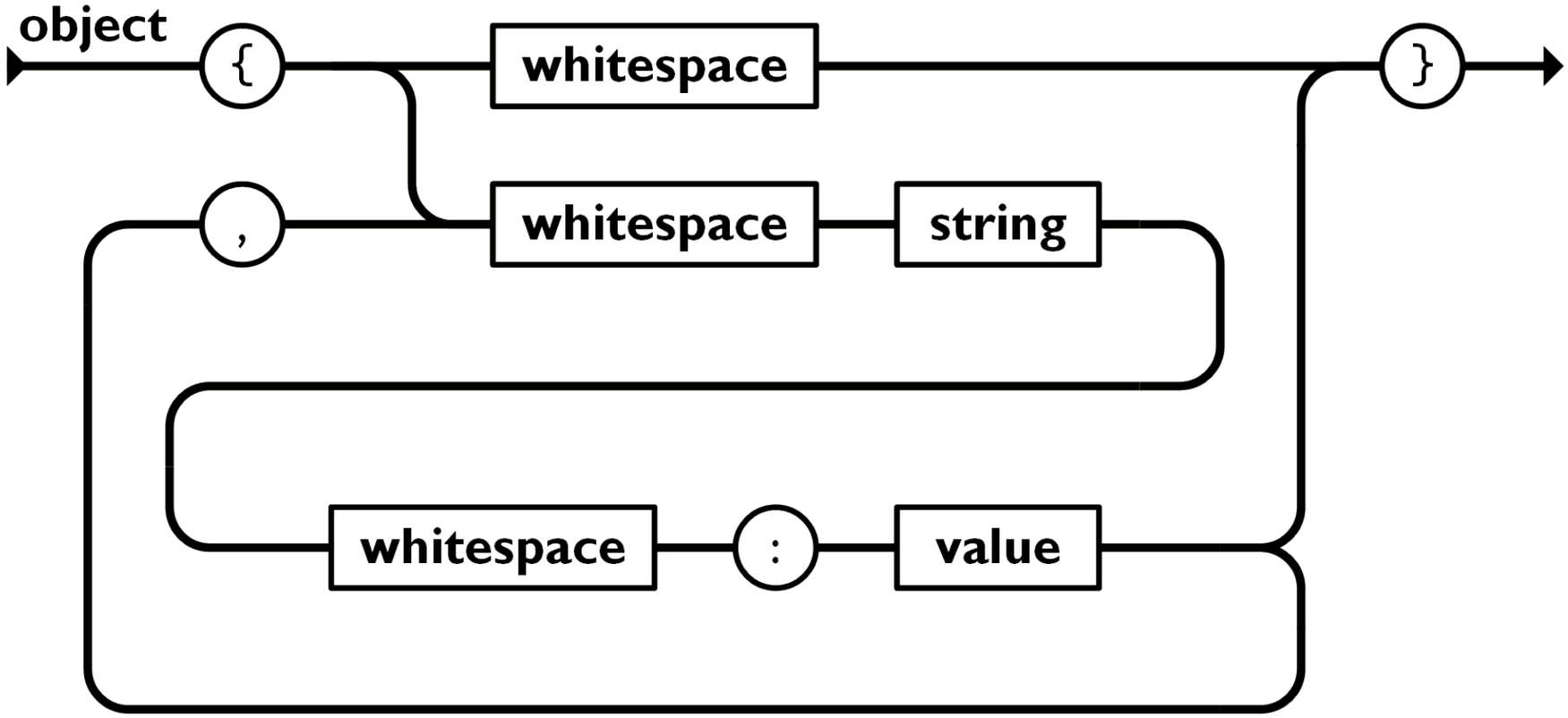
JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

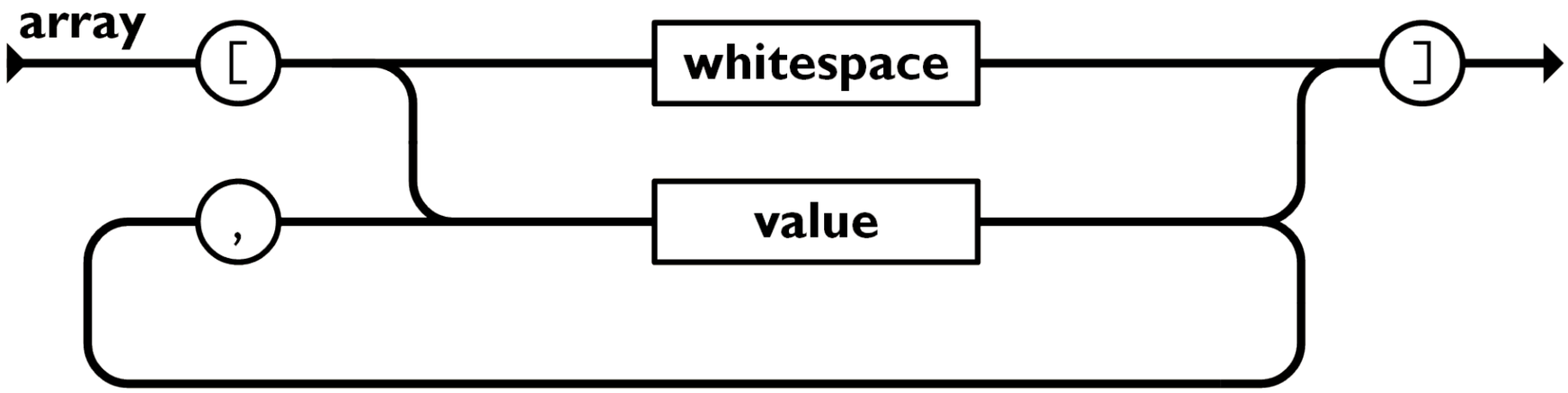
JSON is built on two structures:

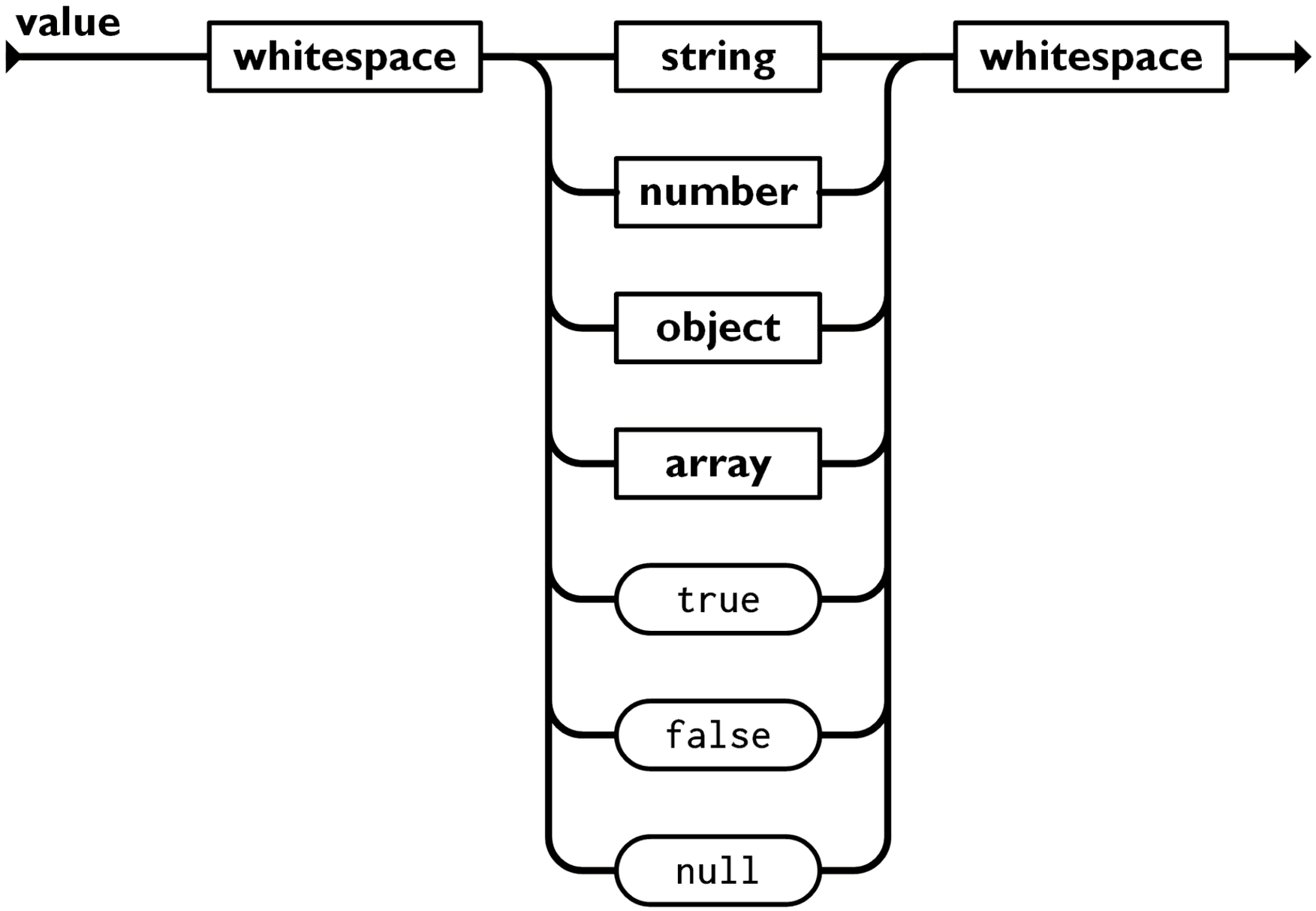
> A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
> An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with

```
json
    element

value
    object
    array
    string
    number
    "true"
    "false"
    "null"

object
    '{' ws '}'
    '{' members '}'
```

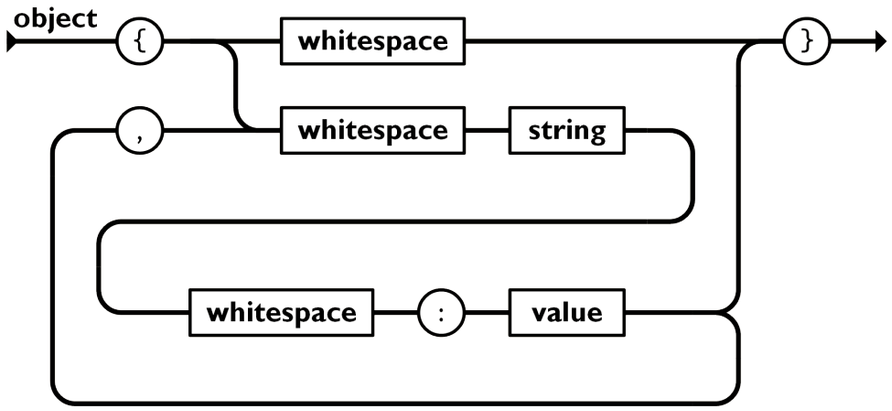

object
{
whitespace
}
,
whitespace
string
whitespace
:
value


array
[
whitespace
]
,
value


value
whitespace
string
number
object
array
true
false
null
whitespace

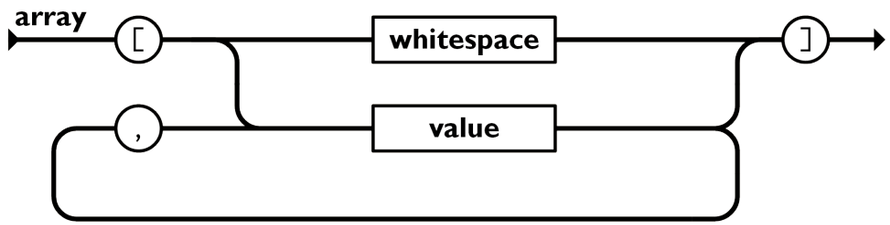
```cpp
struct JsonValue : Custom<JsonValue>
{
  struct result_type;
  static consteval auto get_parser();
};

struct JsonObject : Custom<JsonObject>
{
  using result_type = std::vector<std::tuple<std::string, JsonValue::result_type>>;
  static consteval auto get_parser();
};

struct JsonArray : Custom<JsonArray>
{
  using result_type = std::vector<JsonValue::result_type>;
  static consteval auto get_parser();
};

struct JsonValue::result_type
{
  using variant_t = std::variant<std::string, number_type, JsonObject::result_type,
                                 JsonArray::result_type, bool, nullptr_t>;
  variant_t variant;
};
```

```cpp
consteval auto JsonObject::get_parser()
{
  auto colon = ":"_ign;

  auto pair =
    (ws >> string >> ws >> colon >> JsonValue{});
    // tuple<string, JsonValue::result_type>

  auto object = pair % delimit(","_all);

  auto parser =
    "{"_ign >> (object | ws) >> "}"_ign;
  return parser;
}
```



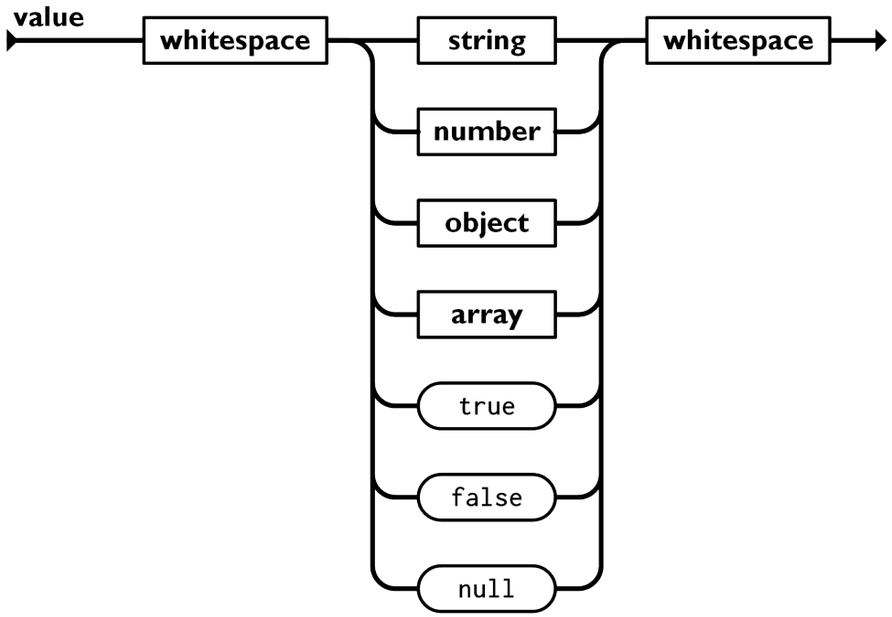
```cpp
struct JsonValue : Custom<JsonValue>
{
  struct result_type;
  static consteval auto get_parser();
};

struct JsonObject : Custom<JsonObject>
{
  using result_type = std::vector<std::tuple<std::string, JsonValue::result_type>>;
  static consteval auto get_parser();
};

struct JsonArray : Custom<JsonArray>
{
  using result_type = std::vector<JsonValue::result_type>;
  static consteval auto get_parser();
};

struct JsonValue::result_type
{
  using variant_t = std::variant<std::string, number_type, JsonObject::result_type,
                                 JsonArray::result_type, bool, nullptr_t>;
  variant_t variant;
};
```

```
1 consteval auto JsonArray::get_parser()
2 {
3   auto array = JsonValue{} % delimit(","_all);
4
5   auto parser =
6     "["_ign >> (array | ws) >> "]"_ign;
7   return parser;
8 }
```



```cpp
struct JsonValue : Custom<JsonValue>
{
  struct result_type;
  static consteval auto get_parser();
};

struct JsonObject : Custom<JsonObject>
{
  using result_type = std::vector<std::tuple<std::string, JsonValue::result_type>>;
  static consteval auto get_parser();
};

struct JsonArray : Custom<JsonArray>
{
  using result_type = std::vector<JsonValue::result_type>;
  static consteval auto get_parser();
};

struct JsonValue::result_type
{
  using variant_t = std::variant<std::string, number_type, JsonObject::result_type,
                                 JsonArray::result_type, bool, nullptr_t>;
  variant_t variant;
};
```


```cpp
consteval auto JsonValue::get_parser()
{
  auto parser = into<result_type>
  (
    string |
    number |
    JsonObject{} |
    JsonArray{} |
    "true"_all  % constant<true> |
    "false"_all % constant<false> |
    "null"_all  % constant<nullptr>
  );

  return ws >> parser >> ws;
}
```



```
auto input = R"-({"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New",
        "onclick": "CreateNewDoc()"},
      {"value": "Open",
        "onclick": "OpenDoc()"},
      {"value": "Close",
        "onclick": "CloseDoc()"}
    ]
  }
}})-";

auto result = JsonObject::parse(input);
```

Unit testing frameworks

Frameworks

- GoogleTest
- Boost.Test
- Catch2
- doctest
- snitch
- UT

Steps

0. Set up a test case, same for each framework
1. Basic run-time test
2. Custom error messages at run-time
3. Basic compile-time test
4. Custom error messages at compile-time 🤞

Test setup

```
auto digit = "0123456789"_any_of;
auto year = exactly<4>(digit) % join;
auto month = exactly<2>(digit) % join;
auto day = digit % exactly<2> % join;
```

```
// ISO-8601 YYYY-MM-DD
auto parser = year >> "-"_ign >>
              month >> "-"_ign >> day;
```

- Test parse result
- Test combinator operations
- Test parser equality
- Test parser properties

Run-time testing with GoogleTest

```
TEST(ISO8601, parse_result)
{
  auto result = parser.parse("2024-01-01");

  ASSERT_TRUE(result.has_value());
  EXPECT_EQ(*result, std::tuple("2024", "01", "01"));
}
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ISO8601
[ RUN      ] ISO8601.parse_result
[       OK ] ISO8601.parse_result (0 ms)
[----------] 1 test from ISO8601 (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (1 ms total)
[  PASSED  ] 1 test.
```

Run-time testing with GoogleTest

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ISO8601
[ RUN      ] ISO8601.parse_result
path\to\main.cpp(17): error: Value of: result.has_value()
  Actual: false
Expected: true

[  FAILED  ] ISO8601.parse_result (0 ms)
[----------] 1 test from ISO8601 (1 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (5110 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] ISO8601.parse_result

 1 FAILED TEST
```

Run-time testing with GoogleTest

```
TEST(ISO8601, parse_result)
{

  auto result = parser.parse("bad");

  ASSERT_TRUE(result.has_value());


  EXPECT_EQ(*result, std::tuple("2024", "01", "01"));
}
```

Run-time testing with GoogleTest

```
TEST(ISO8601, parse_result)
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  ASSERT_TRUE(result.has_value());


  EXPECT_EQ(*result, std::tuple("2024", "01", "01"));
}
```

Run-time testing with GoogleTest

```
TEST(ISO8601, parse_result)
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  ASSERT_TRUE(result.has_value())
    << std::format("Input could not be parsed.\nInput = \"{}\"\n"
    "Parser = {}", input, k3::tok3n::pretty(p).view());
  EXPECT_EQ(*result, std::tuple("2024", "01", "01"));
}
```

Run-time testing with GoogleTest

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ISO8601
[ RUN      ] ISO8601.parse_result
path\to\main.cpp(19): error: Value of: result.has_value()
  Actual: false
Expected: true
Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>

[...]
```

Run-time testing with GoogleTest

```
auto digit = "0123456789"_any_of;
auto year = exactly<4>(digit) % join;
auto month = exactly<2>(digit) % join;
auto day = digit % exactly<2> % join;

auto parser = year >> "-"_ign >>
              month >> "-"_ign >> day;
```

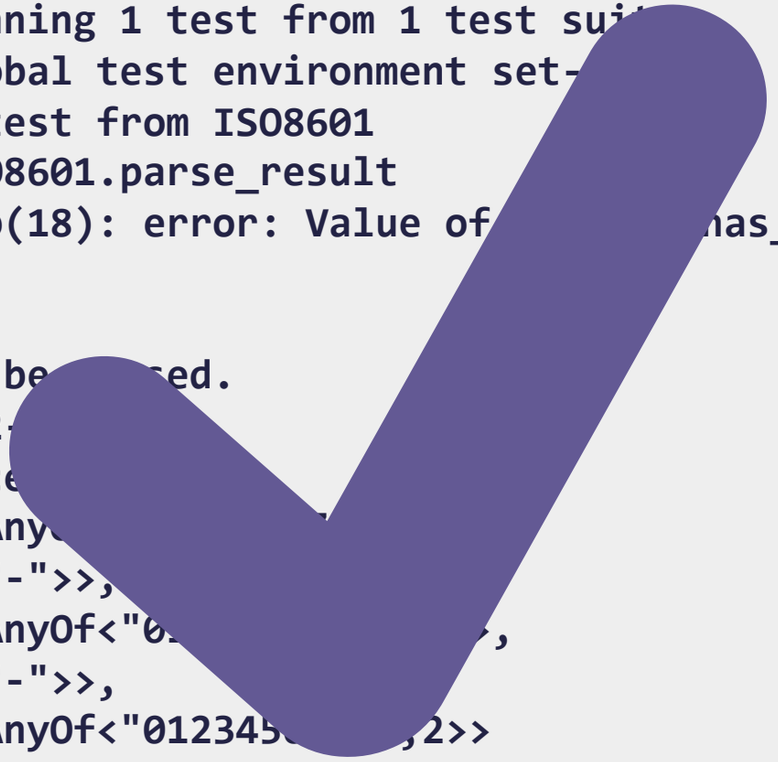
```
TEST(ISO8601, parse_result)
{
  std::string_view input = "2024-02-29";
  auto result = parser.parse(input);

  ASSERT_TRUE(result.has_value());
  EXPECT_EQ(*result, std::tuple("2024", "02", "29"));
}
```

Run-time testing with GoogleTest

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from ISO8601
[ RUN      ] ISO8601.parse_result
path\to\main.cpp(18): error: Value of: result.has_value()
  Actual: false
Expected: true

[...]
```

Run-time testing with GoogleTest

```
[==========] Running 1 test from 1 test sui
[----------] Global test environment set-
[----------] 1 test from ISO8601
[ RUN      ] ISO8601.parse_result
path\to\main.cpp(18): error: Value of          has_value()
  Actual: false
Expected: true
Input could not be      ed.
Input = "2024-02-
Parser = Sequence
  Join<Exactly<Any
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0          >,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"012345      ,2>>
>

[...]
```



Run-time testing with Boost.Test

```
BOOST_AUTO_TEST_CASE(ISO8601)
{
  auto result = parser.parse("2024-01-01");

  BOOST_REQUIRE(result.has_value());
  BOOST_CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
static_assert failed: 'Type has to implement operator<< to be printable'
```

Run-time testing with Boost.Test

```
BOOST_AUTO_TEST_CASE(ISO8601)
{
  auto result = parser.parse("2024-01-01");

  BOOST_REQUIRE(result.has_value());
  BOOST_CHECK((*result == std::tuple("2024", "01", "01")));
}
```

```
Running 1 test case...

*** No errors detected
```

Run-time testing with Boost.Test

```
BOOST_AUTO_TEST_CASE(ISO8601)
{
  auto result = parser.parse("bad");

  BOOST_REQUIRE(result.has_value());
  BOOST_CHECK((*result == std::tuple("2024", "01", "01")));
}
```

```
Running 1 test case...
main.cpp(49): fatal error: in "ISO8601": critical check result.has_value() has failed

*** 1 failure is detected in the test module "TestModule"
```

Run-time testing with Boost.Test

```
BOOST_AUTO_TEST_CASE(ISO8601)
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  BOOST_REQUIRE_MESSAGE(result.has_value(),
    std::format("\nInput could not be parsed.\nInput = \"{}\"\n"
    "Parser = {}", input, k3::tok3n::pretty(parser).view()));
  BOOST_CHECK((*result == std::tuple("2024", "01", "01")));
}
```

Run-time testing with Boost.Test

```
Running 1 test case...
main.cpp(52): fatal error: in "ISO8601":
Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>

*** 1 failure is detected in the test module "TestModule"
```

Run-time testing with Boost.Test

```
Running 1 test case...
main.cpp(52): fatal error: in "ISO860
Input could not be parsed.
Input = "2024-02-29"
Parser = Sequence<
  Join<Exactly<A          12345678"
  Ignore<AllOf<"
  Join<Exactly<A          67
  Ignore<AllOf<"-"
  Join<Exactly<AnyOf<                  >
>

*** 1 failure is detected in the test module "TestModule"
```

Run-time testing with Catch2

```
TEST_CASE("ISO8601")
{
  std::string_view input = "2024-01-01";
  auto result = parser.parse(input);

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
Randomness seeded to: 1038544898
===============================================================================
All tests passed (2 assertions in 1 test case)
```

Run-time testing with Catch2

```
TEST_CASE("ISO8601")
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
[...]

main.cpp(56): FAILED:
  REQUIRE( result.has_value() )
with expansion:
  false

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Run-time testing with Catch2

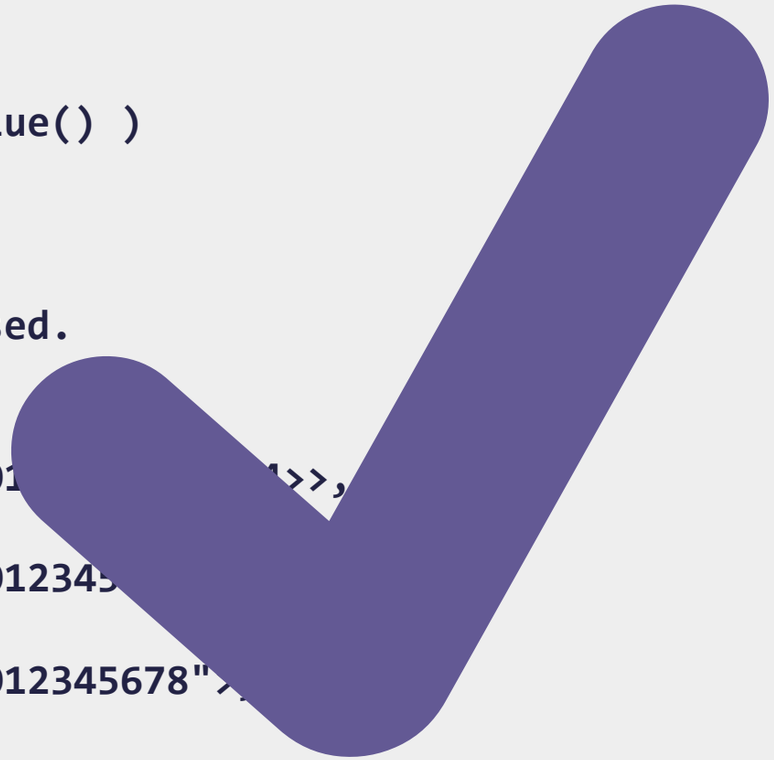
```
TEST_CASE("ISO8601")
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  INFO(std::format("Input could not be parsed.\nInput = \"{}\"\n"
    "Parser = {}", input, k3::tok3n::pretty(parser).view()));

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

Run-time testing with Catch2

```
[...]

main.cpp(59): FAILED:
  REQUIRE( result.has_value() )
with expansion:
  false
with message:
  Input could not be parsed.
  Input = "bad"
  Parser = Sequence<
    Join<Exactly<AnyOf<"0123456789">,4>>,
    Ignore<AllOf<"-">>,
    Join<Exactly<AnyOf<"0123456789">,2>>,
    Ignore<AllOf<"-">>,
    Join<Exactly<AnyOf<"0123456789">,2>>
  >

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Run-time testing with Catch2

```
[...]

main.cpp(59): FAILED:
  REQUIRE( result.has_value() )
with expansion:
  false
with message:
  Input could not be parsed.
  Input = "2024-02-29"
  Parser = Sequence<
    Join<Exactly<AnyOf<"01                1>>,
    Ignore<AllOf<"-">>,
    Join<Exactly<AnyOf<"12345
    Ignore<AllOf<"-">>,
    Join<Exactly<AnyOf<"012345678">
  >

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```



Run-time testing with doctest

```
TEST_CASE("ISO8601")
{
  std::string_view input = "2024-01-01";
  auto result = parser.parse(input);

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
[doctest] doctest version is "2.4.11"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases: 1 | 1 passed | 0 failed | 0 skipped
[doctest] assertions: 2 | 2 passed | 0 failed |
[doctest] Status: SUCCESS!
```

Run-time testing with doctest

```
TEST_CASE("ISO8601")
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
[...]
===============================================================================
\main.cpp(25):
TEST CASE:  ISO8601

\main.cpp(33): FATAL ERROR: REQUIRE( result.has_value() ) is NOT correct!
  values: REQUIRE( false )

===============================================================================
[...]
```

Run-time testing with doctest

```
TEST_CASE("ISO8601")
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  INFO(std::format("Input could not be parsed.\nInput = \"{}\"\n"
    "Parser = {}", input, k3::tok3n::pretty(parser).view()));

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

Run-time testing with doctest

```
[...]
===============================================================================
main.cpp(25):
TEST CASE:  ISO8601

main.cpp(33): FATAL ERROR: REQUIRE( result.has_value() ) is NOT correct!
  values: REQUIRE( false )
  logged: Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>

===============================================================================
[...]
```

Run-time testing with doctest

```
[...]
===============================================================================
main.cpp(25):
TEST CASE:  ISO8601

main.cpp(33): FATAL ERROR: REQUIRE( result.has          ) is NOT correct!
  values: REQUIRE( false )
  logged: Input could not be parsed.
Input = "2024-02-29"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"01234567
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"012345678">,2>
>

===============================================================================
[...]
```

Run-time testing with snitch

```
TEST_CASE("ISO8601")
{
  std::string_view input = "2024-01-01";
  auto result = parser.parse(input);

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
=============================================
success: all tests passed (1 test cases, 3 assertions, 3.830000e-05 seconds)
```

Run-time testing with snitch

```
TEST_CASE("ISO8601")
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

```
starting dev_snitch with snitch v1.2.4.2f62308
=========================================
failed: running test case "ISO8601"
      at main.cpp:21
      REQUIRE(result.has_value()), got: false
=========================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.359400e-03 seconds)
```

Run-time testing with snitch

```
TEST_CASE("ISO8601")
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  INFO(std::format("Input could not be parsed.\nInput = \"{}\"\n"
    "Parser = {}", input, k3::tok3n::pretty(parser).view()));

  REQUIRE(result.has_value());
  CHECK(*result == std::tuple("2024", "01", "01"));
}
```

Run-time testing with snitch

```
starting dev_snitch with snitch v1.2.4.2f62308
===========================================
failed: running test case "ISO8601"
      at main.cpp:33
      with Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>
      REQUIRE(result.has_value()), got: false
===========================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.729600e-03 seconds)
```

Run-time testing with snitch

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "ISO8601"
      at main.cpp:33
      with Input could not be parsed.
Input = "2024-02-29"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123                4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"01234
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"012345678"
>
      REQUIRE(result.has_value()), go       lse
=============================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.729600e-03 seconds)
```

Run-time testing with UT

```
"ISO8601"_test = []
{
  std::string_view input = "2024-01-01";
  auto result = parser.parse(input);

  expect(result.has_value()) << fatal;
  expect(*result == std::tuple("2024", "01", "01"));
};
```

```
Suite 'global': all tests passed (2 asserts in 1 tests)
```

Run-time testing with UT

```
"ISO8601"_test = []
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  expect(result.has_value()) << fatal;
  expect(*result == std::tuple("2024", "01", "01"));
};
```

```
Running test "ISO8601"... FAILED
in: :0 - test condition:  [false]


================================================================================
Suite globaltests:   1 | 2 failed
asserts: 2 | 0 passed | 2 failed
```

Run-time testing with UT

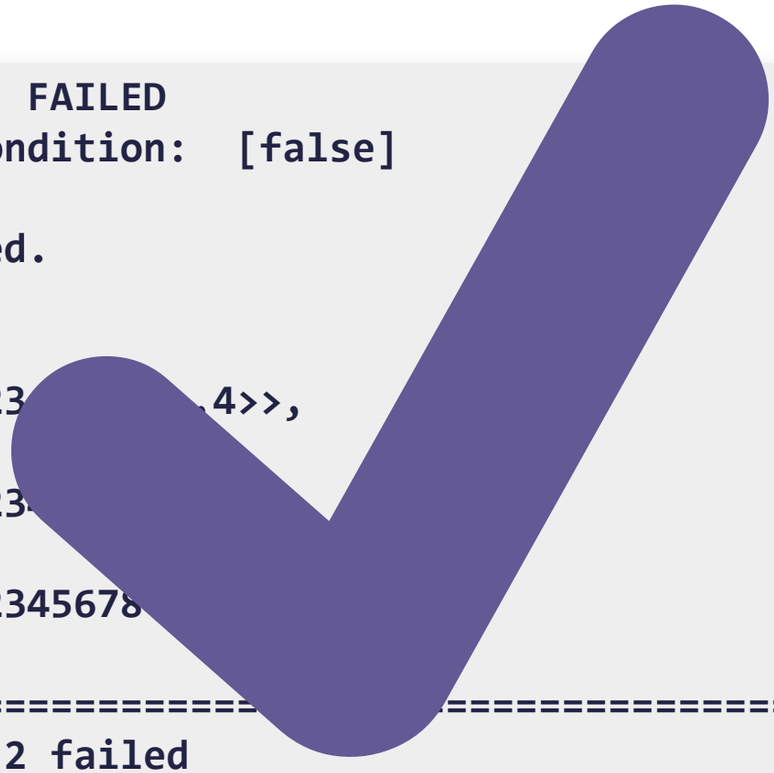
```
"ISO8601"_test = []
{
  std::string_view input = "bad";
  auto result = parser.parse(input);

  expect(result.has_value())
    << std::format("Input could not be parsed.\nInput = \"{}\"\n"
       "Parser = {}", input, k3::tok3n::pretty(parser).view())
    << fatal;
  expect(*result == std::tuple("2024", "01", "01"));
};
```

Run-time testing with UT

```
Running test "ISO8601"... FAILED
in: main.cpp:34 - test condition:  [false]

 Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>
================================================================================
Suite globaltests:   1 | 2 failed
asserts: 2 | 0 passed | 2 failed
```

Run-time testing with UT

```
Running test "ISO8601"... FAILED
in: main.cpp:34 - test condition:  [false]

 Input could not be parsed.
Input = "2024-02-29"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123          4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"012345678
>
==========================================                ======================================
Suite globaltests:   1 | 2 failed
asserts: 2 | 0 passed | 2 failed
```



Compile-time testing

- Level 1 - One at a time
- Level 2 - Parametrized

Compile-time testing

```
auto digit = "0123456789"_any_of;
auto year = exactly<4>(digit) % join;
auto month = exactly<2>(digit) % join;
auto day = digit % exactly<2> % join;
auto parser = year >> "-"_ign >>
              month >> "-"_ign >> day;
```

```
// Test properties like `value_type`, `ParserFor<P, V>`, `family`
```

```
// Test which of these parsers are equal to each other
```

```
// Test combinator operations on the parsers
```

```
// Test the parsing
```

Compile-time testing

```
using Digit   = decltype(digit);
using Year    = decltype(year);
using Month   = decltype(month);
using Day     = decltype(day);
using ISODate = decltype(parser);
```

Compile-time testing with GoogleTest

```
TEST(ISO8601, value_type)
{
  ::testing::StaticAssertTypeEq<Digit::value_type, char>();
  ::testing::StaticAssertTypeEq<Year::value_type, char>();
  ::testing::StaticAssertTypeEq<Month::value_type, char>();
  ::testing::StaticAssertTypeEq<Day::value_type, int>();
  ::testing::StaticAssertTypeEq<ISODate::value_type, char>();
}
```

```
gtest/gtest.h(2156,17): error C2338: static_assert failed:
  'T1 and T2 are not the same type'
main.cpp(33,13): message : see reference to function template instantiation
  'bool testing::StaticAssertTypeEq<char,int>(void) noexcept' being compiled
```

Compile-time testing with GoogleTest

```
TEST(ISO8601, value_type)
{
  static_assert(std::same_as<Digit::value_type, char>);
  static_assert(std::same_as<Year::value_type, char>);
  static_assert(std::same_as<Month::value_type, char>);
  static_assert(std::same_as<Day::value_type, int>);
  static_assert(std::same_as<ISODate::value_type, char>);
}
```

```
main.cpp(33,21): error C2607: static assertion failed
main.cpp(33,21): message : the concept 'std::same_as<char,int>' evaluated to false
concepts(36,9): message : 'char' and 'int' are different types
```

Compile-time testing with GoogleTest

```
TEST(ISO8601, equality)
{
  static_assert(std::same_as<Month, Day>);
  static_assert(!std::same_as<Year, Day>);
}
```

Compile-time testing with GoogleTest

```
TEST(ISO8601, combinators)
{
  static_assert(std::same_as<decltype(Day{} >> Month{}),
                Sequence<Day, Month>>);
}
```

Compile-time testing with GoogleTest

```
TEST(ISO8601, full_date)
{
  static_assert(Parser<ISODate>);
  static_assert(ISODate::family == ParserFamily::Sequence);
  static_assert(ParserFor<ISODate, char>);

  static_assert(std::same_as<ISODate::result_for<char>,
                std::tuple<Output<char>, Output<char>, Output<char>>>);

  static_assert(ISODate::parse("2024-01-01").has_value());
  static_assert(*ISODate::parse("2024-01-01")
                == std::tuple("2024", "01", "01"));
  static_assert(!ISODate::parse("bad").has_value());
}
```

Compile-time testing with GoogleTest

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT) \
  static_assert(P::parse(INPUT).has_value());  \
  ASSERT_TRUE(P::parse(INPUT).has_value());    \
  static_assert(*P::parse(INPUT) == (RESULT)); \
  EXPECT_EQ(*P::parse(INPUT), (RESULT))
```

```
TEST(ISO8601, full_date)
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "01"));
}
```

Compile-time testing with GoogleTest

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT) \
  static_assert(P::parse(INPUT).has_value());  \
  ASSERT_TRUE(P::parse(INPUT).has_value());    \
  static_assert(*P::parse(INPUT) == (RESULT)); \
  EXPECT_EQ(*P::parse(INPUT), (RESULT))
```

```
TEST(ISO8601, full_date)
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
}
```

```
main.cpp(55,2): error C2607: static assertion failed
main.cpp(55,2): error C2131: expression did not evaluate to a constant
optional(200,9): message : failure was caused by call of
  undefined function or one not declared 'constexpr'
optional(200,9): message : see usage of '_CrtDbgReport'
```

Compile-time testing with GoogleTest?

```
#define ASSERT_PARSE_SU      , INPUT, RESULT) \
  static_assert(P::pars      has_value());
  ASSERT_TRUE(P::parse(I      value());
  static_assert(*P::parse(I      (RESU
  EXPECT_EQ(*P::parse(INPUT),      )
```

```
TEST(ISO8601, full_date)
{
  ASSERT_PARSE_SUCCESS(ISODate      le("b", "a", "d"));
}
```

```
main.cpp(55,2): error C      tic assertion
main.cpp(55,2): error (      xpression did not      e to a constant
optional(200,9): messag      ailure was caused by      of
  undefined function or one not declared 'constexpr'
optional(200,9): message : see usage of '_CrtDbgReport'
```

Compile-time testing with Boost.Test

```
BOOST_AUTO_TEST_CASE(ISO8601_properties)
{
  static_assert(Parser<ISODate>);
  static_assert(ISODate::family == ParserFamily::Sequence);
  static_assert(ParserFor<ISODate, char>);
}
```

Compile-time testing with Boost.Test

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT) \
  static_assert(P::parse(INPUT).has_value());  \
  BOOST_REQUIRE(P::parse(INPUT).has_value());  \
  static_assert(*P::parse(INPUT) == (RESULT)); \
  BOOST_CHECK((*P::parse(INPUT) == (RESULT)))
```

```
BOOST_AUTO_TEST_CASE(ISO8601_parsing)
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "01"));
}
```

Compile-time testing with Boost.Test

```
#define ASSERT_PARSE_SU      , INPUT, RESULT)
  static_assert(P::pars      has_value());
  BOOST_REQUIRE(P::parse(    s_value());
  static_assert(*P::parse(I      (RESU
  BOOST_CHECK((*P::parse(INPU
```

```
BOOST_AUTO_TEST_CASE(ISO8601_pars
{
  ASSERT_PARSE_SUCCESS(ISODate      le("b", "a", "d"));
}
```

```
main.cpp(62,2): error C      tic assertion
main.cpp(62,2): error (      xpression did not      e to a constant
optional(200,9): messag      ailure was caused by      of
  undefined function or one not declared 'constexpr'
optional(200,9): message : see usage of '_CrtDbgReport'
```

Compile-time testing with Catch2

```
// #define CATCH_CONFIG_RUNTIME_STATIC_REQUIRE
#include <catch2/catch_test_macros.hpp>
#include <catch2/catch_session.hpp>
```

```
TEST_CASE("ISO8601 properties")
{
  STATIC_REQUIRE(Parser<ISODate>);
  STATIC_REQUIRE(ISODate::family == ParserFamily::Sequence);
  STATIC_REQUIRE(ParserFor<ISODate, char>);
}
```

```
Randomness seeded to: 391083004
===============================================================================
All tests passed (3 assertions in 1 test case)
```

Compile-time testing with Catch2

```
TEST_CASE("ISO8601 parsing")
{
  STATIC_REQUIRE(ISODate::parse("2024-01-01").has_value());
  STATIC_CHECK(*ISODate::parse("2024-01-01") == std::tuple("2024", "01", "01"));
}
```

Compile-time testing with Catch2

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT) \
  STATIC_REQUIRE(P::parse(INPUT).has_value()); \
  STATIC_CHECK(*P::parse(INPUT) == (RESULT))
```

```
TEST_CASE("ISO8601 parsing")
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "01"));
}
```

Compile-time testing with Catch2

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT) \
  STATIC_REQUIRE(P::parse(INPUT).has_value()); \
  STATIC_CHECK(*P::parse(INPUT) == (RESULT))
```

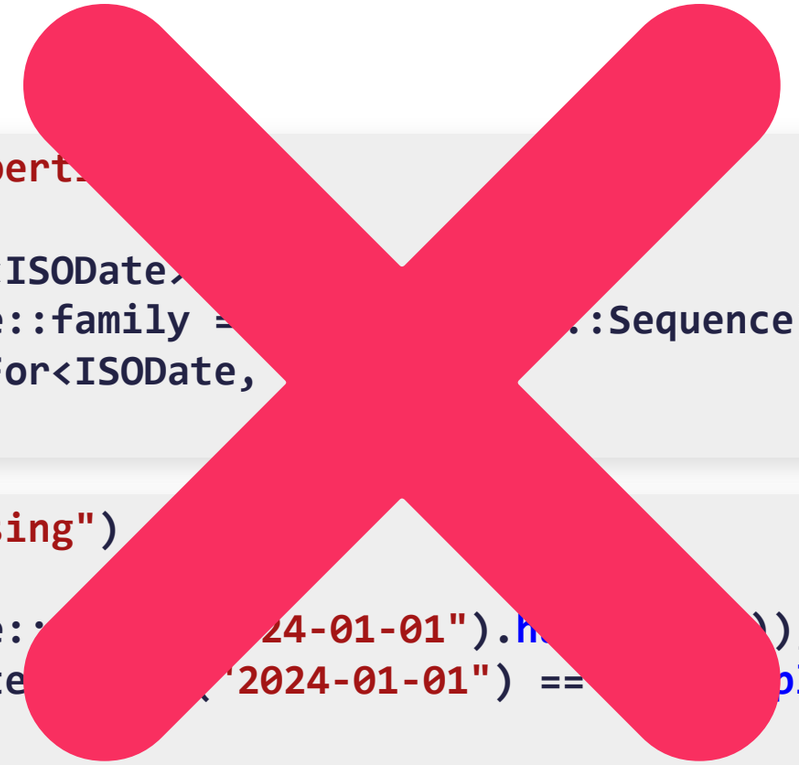
```
TEST_CASE("ISO8601 parsing")
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
}
```

```
main.cpp(92,2): error C2338: static_assert failed: 'ISODate::parse("bad").has_value()'
main.cpp(92,2): error C2131: expression did not evaluate to a constant
optional(200,9): message : failure was caused by call of
  undefined function or one not declared 'constexpr'
optional(200,9): message : see usage of '_CrtDbgReport'
```

Compile-time testing with doctest

```
TEST_CASE("ISO8601 properti
{
  static_assert(Parser<ISODate>
  static_assert(ISODate::family =                    ::Sequence);
  static_assert(ParserFor<ISODate,
}
```

```
TEST_CASE("ISO8601 parsing")
{
  static_assert(ISODate::          24-01-01").h          ));
  static_assert(*ISODate           "2024-01-01") ==          ble("2024", "01", "01"));
}
```



Compile-time testing with snitch

```
TEST_CASE("ISO8601 properties")
{
  CONSTEVAL_REQUIRE(Parser<ISODate>);
  CONSTEVAL_REQUIRE(ISODate::family == ParserFamily::Sequence);
  CONSTEVAL_REQUIRE(ParserFor<ISODate, char>);
}
```

```
starting dev_snitch with snitch v1.2.4.2f62308
===========================================
===========================================
success: all tests passed (1 test cases, 4 assertions, 4.970000e-05 seconds)
```

Compile-time testing with snitch

```
TEST_CASE("ISO8601 parsing")
{
  CONSTEXPR_REQUIRE(ISODate::parse("2024-01-01").has_value());
  CONSTEXPR_CHECK(*ISODate::parse("2024-01-01") == std::tuple("2024", "01", "01"));
}
```

Compile-time testing with snitch

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)    \
  CONSTEXPR_REQUIRE(P::parse(INPUT).has_value()); \
  CONSTEXPR_CHECK(*P::parse(INPUT) == (RESULT))
```

```
TEST_CASE("ISO8601 parsing")
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "01"));
}
```

```
starting dev_snitch with snitch v1.2.4.2f62308
===============================================
===============================================
success: all tests passed (1 test cases, 5 assertions, 7.280000e-05 seconds)
```

Compile-time testing with snitch

```
TEST_CASE("ISO8601 parsing")
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "02"));
}
```

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "ISO8601 parsing"
      at main.cpp:66
      CONSTEXPR_CHECK[compile-time]
        (*ISODate::parse("2024-01-01") == (std::tuple("2024", "01", "02")))
      got: ? != ?
failed: running test case "ISO8601 parsing"
      at main.cpp:66
      CONSTEXPR_CHECK[run-time]
        (*ISODate::parse("2024-01-01") == (std::tuple("2024", "01", "02")))
      got: ? != ?
=============================================
error: all tests failed (1 out of 1 test cases, 5 assertions, 7.613000e-04 seconds)
```

Compile-time testing with snitch

```
TEST_CASE("ISO8601 parsing")
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
}
```

```
main.cpp(89,5): error C2131: expression did not evaluate to a constant
optional(200,9): message : failure was caused by call of undefined function or one not declared 'constexpr'
optional(200,9): message : see usage of '_CrtDbgReport'
main.cpp(89,5): message : the call stack of the evaluation (the oldest call first) is
main.cpp(89,5): message : while evaluating function 'T &&k3::tok3n::Result<T,char>::operator *(void) &&'
    with
    [
        T=std::tuple<k3::tok3n::Output<char>,k3::tok3n::Output<char>,k3::tok3n::Output<char>>
    ]
Result.h(34,51): message : while evaluating function '_Ty &&std::_Optional_construct_base<_Ty>::operator *(void) noexcept &&'
    with
    [
        _Ty=std::tuple<k3::tok3n::Output<char>,k3::tok3n::Output<char>,k3::tok3n::Output<char>>
    ]
optional(200,9): message : while evaluating function 'int _CrtDbgReport(int,const char *,int,const char *,const char *,...)'
```

Compile-time testing with snitch

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)      \
  CONSTEXPR_REQUIRE(P::parse(INPUT).has_value());   \
  if constexpr (P::parse(INPUT).has_value()) {      \
    CONSTEXPR_CHECK(*P::parse(INPUT) == (RESULT));  \
  } else {                                          \
    CHECK(*P::parse(INPUT) == (RESULT));            \
  }
```

```
main.cpp(89,5): error C2131: expression did not evaluate to a constant
optional(200,9): message : failure was caused by call of undefined function or one not declared 'constexpr'
optional(200,9): message : see usage of '_CrtDbgReport'
main.cpp(89,5): message : the call stack of the evaluation (the oldest call first) is
main.cpp(89,5): message : while evaluating function 'T &&k3::tok3n::Result<T,char>::operator *(void) &&'
    with
    [
        T=std::tuple<k3::tok3n::Output<char>,k3::tok3n::Output<char>,k3::tok3n::Output<char>>
    ]
[...]
```

Compile-time testing with snitch

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)       \
  [](auto) {                                         \
    CONSTEXPR_REQUIRE(P::parse(INPUT).has_value());  \
    if constexpr (P::parse(INPUT).has_value())  {    \
      CONSTEXPR_CHECK(*P::parse(INPUT) == (RESULT)); \
    } else {                                         \
      CHECK(*P::parse(INPUT) == (RESULT));           \
    }                                                \
  }(int{})
```

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "ISO8601 parsing"
      at main.cpp:125
      CONSTEXPR_REQUIRE[compile-time](ISODate::parse("bad").has_value())
      got: false
=============================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.432300e-03 seconds)
```

Compile-time testing with Catch2

```
#define ASSERT_PARSE_SUCCESS(P                RESULT)     \
  [](auto) {
    STATIC_REQUIRE(P::parse(INPUT)                 ()
    if constexpr (P::parse(INPUT).ha
      STATIC_CHECK(*P::parse(INPUT) ==                    \
    } else {                                              \
      CHECK(*P::parse(INPUT) == (RESU
    }
  }(int{})
```

```
main.cpp(106,5): error C2338:          ssert failed:          :parse("bad").has_value()'
main.cpp(106,5): message : Th          gnostic occurred in          ompiler generated
  function 'auto CATCH2_INTERNAL_TEST_0::<lambda_1>::operator ()(_T1) const'
```

Compile-time testing with snitch

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)       \
  [](auto) {                                         \
    CONSTEXPR_REQUIRE(P::parse(INPUT).has_value());  \
    if constexpr (P::parse(INPUT).has_value())  {    \
      CONSTEXPR_CHECK(*P::parse(INPUT) == (RESULT)); \
    } else {                                         \
      CHECK(*P::parse(INPUT) == (RESULT));           \
    }                                                \
  }(int{})
```

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "ISO8601 parsing"
      at main.cpp:125
      CONSTEXPR_REQUIRE[compile-time](ISODate::parse("bad").has_value())
      got: false
=============================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.432300e-03 seconds)
```

Compile-time testing with snitch

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)                          \
  [](auto) {                                                            \
    {                                                                   \
      INFO(std::format("Input could not be parsed.\nInput = \"{}\"\n"   \
        "Parser = {}", (INPUT), k3::tok3n::pretty(P{}).view()));        \
      CONSTEXPR_REQUIRE(P::parse(INPUT).has_value());                   \
    }                                                                   \
    if constexpr (P::parse(INPUT).has_value())  {                       \
      CONSTEXPR_CHECK(*P::parse(INPUT) == (RESULT));                    \
    } else {                                                            \
      CHECK(*P::parse(INPUT) == (RESULT));                              \
    }                                                                   \
  }(int{})
```

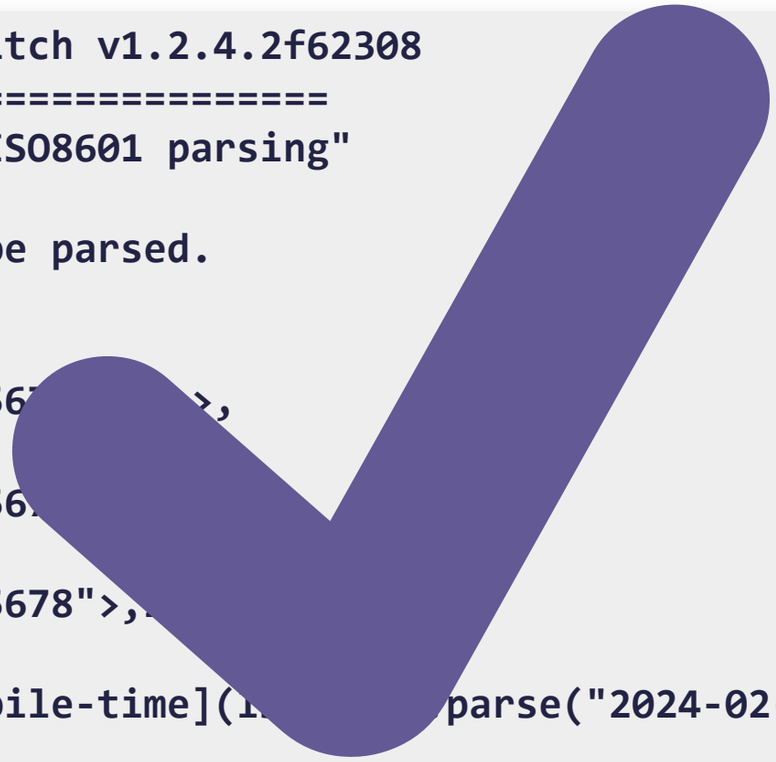
```
TEST_CASE("ISO8601 parsing")
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
}
```

Compile-time testing with snitch

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "ISO8601 parsing"
      at main.cpp:125
      with Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>
      CONSTEXPR_REQUIRE[compile-time](ISODate::parse("bad").has_value())
      got: false
=============================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.862600e-03 seconds)
```

Compile-time testing with snitch

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "ISO8601 parsing"
      at main.cpp:125
      with Input could not be parsed.
Input = "2024-02-29"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456      >>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"012345678">>,
>
      CONSTEXPR_REQUIRE[compile-time](i        parse("2024-02-29").has_value())
      got: false
=============================================
error: all tests failed (1 out of 1 test cases, 1 assertions, 1.862600e-03 seconds)
```



Compile-time testing with UT

```
"ISO8601 properties"_test = []
{
  expect(constant<Parser<ISODate>>);
  expect(constant<ISODate::family == ParserFamily::Sequence>);
  expect(constant<ParserFor<ISODate, char>>);
};
```

```
Suite 'global': all tests passed (3 asserts in 1 tests)
```

Compile-time testing with UT

```
"ISO8601 parsing"_test = []
{
  expect(constant<ISODate::parse("2024-01-01").has_value()>) << fatal;
  expect(constant<*ISODate::parse("2024-01-01") == std::tuple("2024", "01", "01")>);
};
```

Compile-time testing with UT

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)    \
  expect(constant<P::parse(INPUT).has_value()>);  \
  expect(P::parse(INPUT).has_value()) << fatal;   \
  expect(constant<*P::parse(INPUT) == (RESULT)>); \
  expect(*P::parse(INPUT) == (RESULT))
```

```
"ISO8601 parsing"_test = []
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "01"));
};
```

```
Suite 'global': all tests passed (4 asserts in 1 tests)
```

Compile-time testing with UT

```
"ISO8601 parsing"_test = []
{
  ASSERT_PARSE_SUCCESS(ISODate, "2024-01-01", std::tuple("2024", "01", "02"));
};
```

```
Running test "ISO8601 parsing"... FAILED
in: main.cpp:76 - test condition:  [false]

Running test "ISO8601 parsing"... FAILED
in: main.cpp:76 - test condition:  [false]FAILED
in: main.cpp:76 - test condition:  [false]

================================================================================
Suite globaltests:   1 | 2 failed
asserts: 4 | 0 passed | 2 failed
```

Compile-time testing with UT

```
"ISO8601 parsing"_test = []
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
};
```

```
main.cpp(77,3): error C2975: 'Constant': invalid template argument for
  'boost::ext::ut::v2_0_1::constant', expected compile-time constant expression
ut.hpp(2982,11): message : see declaration of 'Constant'
```

Compile-time testing with UT

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)             \
  expect(constant<P::parse(INPUT).has_value()>) << fatal; \
  expect(P::parse(INPUT).has_value()) << fatal;            \
  if constexpr (P::parse(INPUT).has_value()) {             \
    expect(constant<*P::parse(INPUT) == (RESULT)>);        \
  }                                                        \
  expect(*P::parse(INPUT) == (RESULT))
```

```
"ISO8601 parsing"_test = []
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
};
```

```
Running test "ISO8601 parsing"... FAILED
[...]

================================================================================
Suite globaltests:   1 | 3 failed
asserts: 3 | 0 passed | 3 failed
```

Compile-time testing with UT

```
#define EXPECT_CT_AND_RT_FATAL(COND, MESSAGE)     \
  expect(constant<(COND)>) << (MESSAGE) << fatal; \
  expect(COND) << (MESSAGE) << fatal

#define EXPECT_CT_AND_RT_NONFATAL(COND) \
  expect(constant<(COND)>);             \
  expect(COND)

#define ASSERT_PARSE_SUCCESS(P, INPUT, RESULT)                  \
  EXPECT_CT_AND_RT_FATAL(P::parse(INPUT).has_value(),           \
    std::format("Input could not be parsed.\nInput = \"{}\"\n" \
    "Parser = {}", (INPUT), k3::tok3n::pretty(P{}).view()));    \
  if constexpr (P::parse(INPUT).has_value()) {                  \
    EXPECT_CT_AND_RT_NONFATAL(*P::parse(INPUT) == (RESULT));    \
  }
```

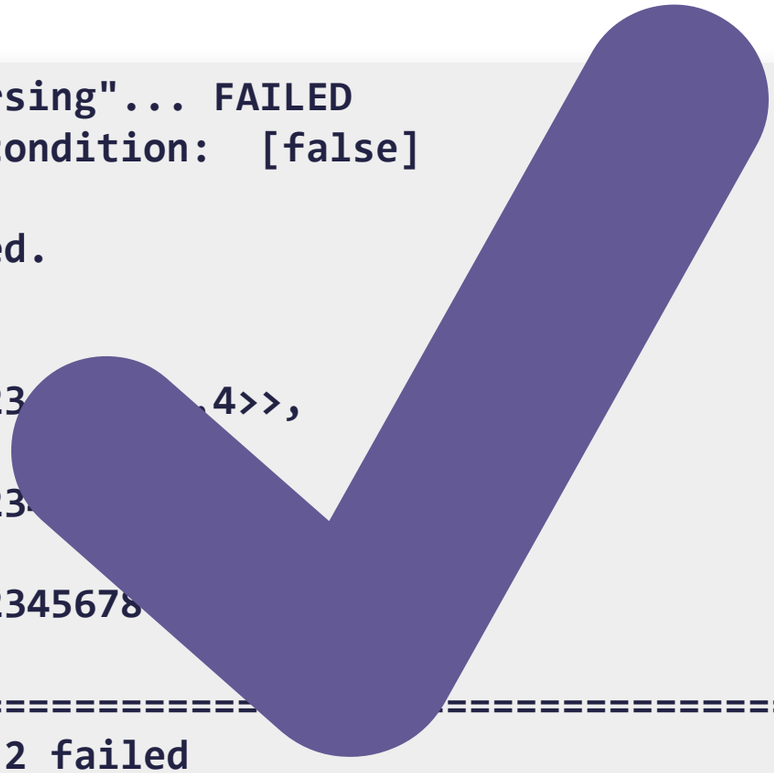
Compile-time testing with UT

```
"ISO8601 parsing"_test = []
{
  ASSERT_PARSE_SUCCESS(ISODate, "bad", std::tuple("b", "a", "d"));
};
```

```
Running test "ISO8601 parsing"... FAILED
in: main.cpp:103 - test condition:  [false]

 Input could not be parsed.
Input = "bad"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123456789">,4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123456789">,2>>
>
================================================================================
Suite globaltests:   1 | 2 failed
asserts: 2 | 0 passed | 2 failed
```

Compile-time testing with UT

```
Running test "ISO8601 parsing"... FAILED
in: main.cpp:103 - test condition:  [false]

 Input could not be parsed.
Input = "2024-02-29"
Parser = Sequence<
  Join<Exactly<AnyOf<"0123      4>>,
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"0123
  Ignore<AllOf<"-">>,
  Join<Exactly<AnyOf<"012345678
>
==========================================      ==========================================
Suite globaltests:   1 | 2 failed
asserts: 2 | 0 passed | 2 failed
```



Parametrized compile-time testing with snitch

```
template <class L, class R>
concept can_sequence = requires { L{} >> R{}; };
```

```
template <Parser L, Parser R>
void sequence_tester(L, R)
{
  if (std::same_as<typename L::value_type, typename R::value_type>)
  {
    INFO(std::format("`L >> R` did not compile.\nL = {}\nR = {}",
      k3::tok3n::pretty(L{}).view(), k3::tok3n::pretty(R{}).view()));
    CONSTEXPR_CHECK(can_sequence<L, R>);
  }
  else
  {
    INFO(std::format("`L >> R` compiled but it shouldn't.\nL = {}\nR = {}",
      k3::tok3n::pretty(L{}).view(), k3::tok3n::pretty(R{}).view()));
    CONSTEXPR_CHECK(!can_sequence<L, R>);
  }
}
```

Parametrized compile-time testing with snitch

```
template <Parser... Ps>
void pairwise_sequence_tester(type_list<Ps...>)
{
  constexpr auto impl = []<Parser R>(R)
  {
    (..., sequence_tester(Ps{}, R{}));
  };
  (..., impl(Ps{}));
}
```

```
TEST_CASE("Combinable with sequence")
{
  using P1 = decltype("123"_any_of); // value_type is char
  using P2 = decltype("456"_all_of);
  using P3 = decltype(L"123"_any_of); // value_type is wchar_t
  using P4 = decltype(L"456"_all_of);
  using List = type_list<P1, P2, P3, P4>;
  pairwise_sequence_tester(List{});
}
```

Parametrized compile-time testing with snitch

```
starting dev_snitch with snitch v1.2.4.2f62308
=============================================
failed: running test case "Combinable with sequence"
      at main.cpp:104
      with `L >> R` did not compile.
L = AnyOf<"123">
R = AnyOf<"456">
      CONSTEXPR_CHECK[compile-time](can_sequence<L, R>)
      got: false
failed: running test case "Combinable with sequence"
      at main.cpp:110
      with `L >> R` compiled but it shouldn't.
L = AnyOf<"123">
R = AnyOf<L"123">
      CONSTEXPR_CHECK[compile-time](!can_sequence<L, R>)
      got: false
...
```

My solution

My prior assumptions

- Compile-time test failure must fail the compile
- Similarly, error messages must be in the `static_assert`
- So, working with these assumptions...

`static_assert` messages in C++20

[dcl.pre]

In a *static_assert-declaration*, the *constant-expression* shall be a contextually converted constant expression of type `bool` [...] the resulting diagnostic message (4.1) shall include the text of the *string-literal*, if one is supplied, [...]

Mini testing library

```
#define ASSERT(CONDITION, MESSAGE)                    \
  {                                                   \
    STATIC_ASSERT(CONDITION, MESSAGE);                \
    if (!TestResultContext::check(CONDITION))         \
      return TestResultContext::add_error(MESSAGE); \
  }
```

Mini testing library

```
#ifdef DISABLE_STATIC_ASSERT
#define STATIC_ASSERT(...)
#else
#define STATIC_ASSERT(...) static_assert(__VA_ARGS__)
#endif
```

Mini testing library

```
#define ASSERT(CONDITION, MESSAGE)                    \
  {                                                   \
    STATIC_ASSERT(CONDITION, MESSAGE);                \
    if (!TestResultContext::check(CONDITION))         \
      return TestResultContext::add_error(MESSAGE); \
  }
```

Mini testing library

```
#define ASSERT(CONDITION, MESSAGE)                    \
  {                                                   \
    STATIC_ASSERT(CONDITION, MESSAGE);                \
    if (!TestResultContext::check(CONDITION))         \
      return TestResultContext::add_error(MESSAGE); \
  }
```

```
void TestResultContext::add_error(std::string_view message,
  std::source_location location = std::source_location::current());
```

Simple test

```
FIXTURE("AnyOf");

TEST("AnyOf", "Parse single 'a'")
{
  using P = AnyOf<'a'>;

  ASSERT(Parser<P>, "AnyOf<'a'> does not satisfy the Parser concept");

  ASSERT(P::parse("ab").has_value(),
    R"(`AnyOf<'a'>::parse("ab")` does not give a valid result)");
  ASSERT(*(P::parse("ab")) == "a",
    R"(`*(AnyOf<'a'>::parse("ab"))` does not equal `"a"`)");
  ASSERT(P::parse("ab").remaining() == "b",
    R"(`AnyOf<'a'>::parse("ab").remaining()` does not equal `"b"`)");

  ASSERT(not P::parse("ba").has_value(),
    R"(`AnyOf<'a'>::parse("ba")` gives a valid result but it should not)");
  ASSERT(P::parse("ba").remaining() == "ba",
    R"(`AnyOf<'a'>::parse("ba").remaining()` does not equal `"ba"`)");
}
```

Simple test

```
Displaying all fixture results, 1 total.

Fixture "AnyOf" - 1 tests / 0 failures.
```


Simple test

```
FIXTURE("AnyOf");

TEST("AnyOf", "Parse single 'a'")
{
  using P = AnyOf<'a'>;

  // ...

  ASSERT(P::parse("Ab").has_value(),
    R"(`AnyOf<'a'>::parse("Ab")` does not give a valid result)");
  ASSERT(*(P::parse("Ab")) == "A",
    R"(`*(AnyOf<'a'>::parse("Ab"))` does not equal `"A"`)");

  // ...
}
```

Simple test

```
AnyOf.cpp(38,2): error C2338: static_assert failed:
  ``AnyOf<'a'>::parse("Ab")` does not give a valid result'
```

```
Displaying all fixture results, 1 total.

Fixture "AnyOf" - 1 tests / 1 failures.

=================================

Fixture "AnyOf" - 1 tests
  Test "Parse single 'a'" - 1 checks / 1 errors.

Test "Parse single 'a'" failed at
  File: AnyOf.cpp
  Line: 38
  Message: `AnyOf<'a'>::parse("Ab")` does not give a valid result
```

Infeasable to write many of these

```
FIXTURE("AnyOf");

TEST("AnyOf", "Parse single 'a'")
{
  using P = AnyOf<'a'>;

  ASSERT(Parser<P>, "AnyOf<'a'> does not satisfy the Parser concept");

  ASSERT(P::parse("ab").has_value(),
    R"(`AnyOf<'a'>::parse("ab")` does not give a valid result)");
  ASSERT(*(P::parse("ab")) == "a",
    R"(`*(AnyOf<'a'>::parse("ab"))` does not equal `"a"`)");
  ASSERT(P::parse("ab").remaining() == "b",
    R"(`AnyOf<'a'>::parse("ab").remaining()` does not equal `"b"`)");

  ASSERT(not P::parse("ba").has_value(),
    R"(`AnyOf<'a'>::parse("ba")` gives a valid result but it should not)");
  ASSERT(P::parse("ba").remaining() == "ba",
    R"(`AnyOf<'a'>::parse("ba").remaining()` does not equal `"ba"`)");
}
```

Infeasable to write many of these

```
FIXTURE("AnyOf");

TEST("AnyOf", "Parse single 'a'")
{
  using P =

  ASSERT(Pa     <P>, "A      'a'>     es not     tisfy     P     r conc     ');

  ASSERT(P:     se("ab")     _va
    R"(`Any     a'>::pa     ab")     oes     give a va     result)");
  ASSERT(*(     anse(     ) ==
    R"(`*(A     parse("ab"     does     equal `     `)");
  ASSERT(P::parse("ab").remaining() == "b",
    R"(`AnyOf<'a'>::parse("ab").remaining()` does not equal `"b"`)");

  ASSERT(not P::parse("ba").has_value(),
    R"(`AnyOf<'a'>::parse("ba")` gives a valid result but it should not)");
  ASSERT(P::parse("ba").remaining() == "ba",
    R"(`AnyOf<'a'>::parse("ba").remaining()` does not equal `"ba"`)");
}
```

DRY!

Parametrize the testing

```
#define ASSERT_PARSE_SUCCESS(P, INPUT, OUTPUT, REMAIN)                          \
  ASSERT((Parser<P>),                                                           \
    "`" STR(P) "` does not satisfy the Parser concept.");                       \
  ASSERT(P::parse(INPUT).has_value(),                                           \
    "`" STR(P) "::parse(" STR(INPUT) ")` does not give a valid result" );       \
  ASSERT(*(P::parse(INPUT)) == (OUTPUT),                                        \
    "`*(" STR(P) "::parse(" STR(INPUT) "))` does not equal `" STR(OUTPUT) "`"); \
  ASSERT(P::parse(INPUT).remaining() == (REMAIN),                               \
    "`" STR(P) "::parse(" STR(INPUT) ").remaining()` does not equal `" STR(REMAIN) "`")
```

Parametrize the testing

```
TEST("AnyOf", "Parse single 'a'")
{
  ASSERT_PARSE_SUCCESS(AnyOf<'a'>, "ab", "a", "b");
}
```

```
Displaying all fixture results, 1 total.

Fixture "AnyOf" - 1 tests / 0 failures.
```

Parametrize the testing

```
TEST("AnyOf", "Parse single 'a'")
{
  ASSERT_PARSE_SUCCESS(AnyOf<'a'>, "Ab", "A", "b");
}
```

```
AnyOf.cpp(44,2): error C2338: static_assert failed:
  '`AnyOf<'a'>::parse("Ab")` does not give a valid result'
```

Parametrize the testing

```
TEST("AnyOf", "Parse single 'a'")
{
  ASSERT_PARSE_SUCCESS(AnyOf<'a'>, "ab", "A", "b");
}
```

```
AnyOf.cpp(44,2): error C2338: static_assert failed:
  '`*(AnyOf<'a'>::parse("ab"))` does not equal `"A"`'
```

Parametrize the testing

```
TEST("AnyOf", "Parse single 'a'")
{
  ASSERT_PARSE_SUCCESS(AnyOf<'a'>, "ab", "a", "B");
}
```

```
AnyOf.cpp(44,2): error C2338: static_assert failed:
  '`AnyOf<'a'>::parse("ab").remaining()` does not equal `"B"`'
```

Trouble with string literals

```
TEST("ISO-8601", "parse")
{
  auto digit = "0123456789"_any_of;
  auto year = exactly<4>(digit) % join;
  auto month = exactly<2>(digit) % join;
  auto day = digit % exactly<2> % join;
  auto parser = year >> "-"_ign >> month >> "-"_ign >> day;
  using P = decltype(parser);

  ASSERT_PARSE_SUCCESS(P, "2024-05-02", std::tuple("2024", "05", "02"), "");
}
```

```
Displaying all fixture results, 1 total.

Fixture "ISO-8601" - 1 tests / 0 failures.
```

Trouble with string literals

```
TEST("ISO-8601", "parse")
{
  auto digit = "0123456789"_any_of;
  auto year = exactly<4>(digit) % join;
  auto month = exactly<2>(digit) % join;
  auto day = digit % exactly<2> % join;
  auto parser = year >> "-"_ign >> month >> "-"_ign >> day;
  using P = decltype(parser);

  ASSERT_PARSE_SUCCESS(P, "2024-05-02", std::tuple("2024", "01", "01"), "");
}
```

```
AnyOf.cpp(62,2): error C2338: static_assert failed:
  '`*(P::parse("2024-05-02"))` does not equal `std::tuple("2024", "01", "01")`'
```

Trouble with string literals

```
template <class L, class R>
concept can_sequence = requires { L{} >> R{}; };
```

```
template <Parser L, Parser R>
void sequence_tester(L, R)
{
  if constexpr (std::same_as<typename L::value_type, typename R::value_type>)
  {
    // Where do we pretty print L and R??
    ASSERT((can_sequence<L, R>), "`L >> R` did not compile");
  }
  else
  {
    ASSERT((!can_sequence<L, R>), "`L >> R` compiled but it should not");
  }
}
```

Trouble with string literals

```
ExampleFile.cpp(42,1): error C2338: static_assert failed:
  '`L >> R` compiled but it should not'
```

??

Parametrize the testing - more macros

```
#define ALL_SAMPLES     \
  (AnyOf1) (AnyOf2) ... \
  (AllOf1) (AllOf2) ... \
  ...

// Using BOOST_PP_SEQ_FOR_EACH_PRODUCT over ALL_SAMPLES
#define ASSERT_ALL_SAMPLES_2(ASSERTER) ...
```

Parametrize the testing - more macros

```
#define SEQUENCE_OPERATOR_ASSERTER(L, R)                                          \
  {                                                                               \
    if constexpr (std::same_as<typename L::value_type, typename R::value_type>) \
    {                                                                             \
      ASSERT((can_sequence<L, R>),                                                \
        "`" STR(L) " >> " STR(R) "` did not compile");                            \
    }                                                                             \
    else                                                                          \
    {                                                                             \
      ASSERT((!can_sequence<L, R>),                                               \
        "`" STR(L) " >> " STR(R) "` compiled but it should not");                 \
    }                                                                             \
  }

#define ALL_SAMPLES (AnyOf<"abc">) (AnyOf<"xyz">) (AnyOf<L"abc">) (AnyOf<L"xyz">)

TEST("Combinable with sequence", "all samples")
{
  ASSERT_ALL_SAMPLES_2(SEQUENCE_OPERATOR_ASSERTER);
}
```

Parametrize the testing - more macros

```
AnyOf.cpp(100,2): error C2338: static_assert failed:
  '`AnyOf<"abc"> >> AnyOf<"abc">` compiled but it should not'
AnyOf.cpp(100,2): error C2338: static_assert failed:
  '`AnyOf<"abc"> >> AnyOf<"xyz">` compiled but it should not'
AnyOf.cpp(100,2): error C2338: static_assert failed:
  '`AnyOf<"abc"> >> AnyOf<L"abc">` did not compile'
...
```

Parametrize the testing - more macros

```
#define SEQUENCE_OPERATOR_ASSERTER(L, R)                                          \
  {                                                                               \
    if constexpr (std::same_as<typename L::value_type, typename R::value_type>) \
    {                                                                             \
      ASSERT((can_sequence<L, R>),                                                \
        "`" STR(L) " >> " STR(R) "` did not compile");                            \
    }                                                                             \
    else                                                                          \
    {                                                                             \
      ASSERT((!can_sequence<L, R>),                                               \
        "`" STR(L) " >> " STR(R) "` compiled but it should not");                 \
    }                                                                             \
  }

#define ALL_SAMPLES (AnyOf<"abc">) (AnyOf<"xyz">) (AnyOf<L"abc">) (AnyOf<L"xyz">)

TEST("Combinable with sequence", "all samples")
{
  ASSERT_ALL_SAMPLES_2(SEQUENCE_OPERATOR_ASSERTER);
}
```

Parametrize the testing - more macros

```
#define SEQUENCE_OPERATOR_ASSERTER(L, R)                                          \
  []{                                                                             \
    if constexpr (std::same_as<typename L::value_type, typename R::value_type>) \
    {                                                                             \
      ASSERT((can_sequence<L, R>),                                                \
        "`" STR(L) " >> " STR(R) "` did not compile");                            \
    }                                                                             \
    else                                                                          \
    {                                                                             \
      ASSERT((!can_sequence<L, R>),                                               \
        "`" STR(L) " >> " STR(R) "` compiled but it should not");                 \
    }                                                                             \
  }()

#define ALL_SAMPLES (AnyOf<"abc">) (AnyOf<"xyz">) (AnyOf<L"abc">) (AnyOf<L"xyz">)

TEST("Combinable with sequence", "all samples")
{
  ASSERT_ALL_SAMPLES_2(SEQUENCE_OPERATOR_ASSERTER);
}
```

My framework summary

- Declare with `FIXTURE(name)` and `TEST(fixture_name, name)`
- `ASSERT(cond, msg)` uses both `static_assert` and run-time
- All error messages must be string literals
- Loop over a list of parsers with `ASSERT_ALL_SAMPLES(ASSERTER)` and `ASSERT_ALL_SAMPLES_2(ASSERTER)`
- Result: custom pretty `static_assert` messages

Upcoming C++ features

static_assert(false)

Allowing `static_assert(false)`

Document #:	P2593R1
Date:	2022-11-08
Project:	Programming Language C++
Audience:	CWG

Reply-to: Barry Revzin <barry.revzin@gmail.com>

Contents

§ 1 Revision History

`static_assert(false)`

- Accepted as a DR into C++23 under CWG2518

static_assert(false)

Pre C++23

```
template <class T>
constexpr bool always_false_v = false;
```

```
std::variant<int, string> v = get_variant();

auto visitor = []<class T>(const T& t)
{
  if constexpr (std::same_as<T, int>)
    return std::to_string(t);
  else if constexpr (std::same_as<T,string>)
    return t;
  else
    static_assert(always_false_v<T>);
};

string result = std::visit(visitor, v);
```

C++23

```
//
//
```

```
std::variant<int, string> v = get_variant();

auto visitor = []<class T>(const T& t)
{
  if constexpr (std::same_as<T, int>)
    return std::to_string(t);
  else if constexpr (std::same_as<T,string>)
    return t;
  else
    static_assert(false);
};

string result = std::visit(visitor, v);
```

static_assert(false)

```
#define SEQUENCE_OPERATOR_ASSERTER(L, R)                                          \
  {                                                                               \
    if constexpr (std::same_as<typename L::value_type, typename R::value_type>) \
    {                                                                             \
      ASSERT((can_sequence<L, R>),                                                \
        "`" STR(L) " >> " STR(R) "` did not compile");                            \
    }                                                                             \
    else                                                                          \
    {                                                                             \
      ASSERT((!can_sequence<L, R>),                                               \
        "`" STR(L) " >> " STR(R) "` compiled but it should not");                 \
    }                                                                             \
  }
```

User-generated `static_assert` messages

user-generated `static_assert` messages

Document #:	P2741R3
Date:	2023-06-16
Programming Language C++	
Audience:	EWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose that `static_assert` should accept user-generated string-like objects as their diagnostic message.

User-generated `static_assert` messages

- Expands what can be used as a `static_assert` message
- `M.size()` convertible to `std::size_t`
- `M.data()` convertible to `const char*`
- Accepted into C++26

User-generated `static_assert` messages

```
template <Parser L, Parser R>
void sequence_tester(L, R)
{
  constexpr auto pretty_L = k3::tok3n::pretty(L{}).view();
  constexpr auto pretty_R = k3::tok3n::pretty(R{}).view();

  if constexpr (std::same_as<typename L::value_type, typename R::value_type>)
  {
    ASSERT((can_sequence<L, R>),
      std::format("`L >> R` did not compile\nL = {}\nR = {}",
        pretty_L, pretty_R));
  }
  else
  {
    ASSERT((!can_sequence<L, R>),
      std::format("`L >> R` compiled but it should not\nL = {}\nR = {}",
        pretty_L, pretty_R));
  }
}
```

User-generated `static_assert` messages

```
template <Parser L, Parser R>
void sequence_tester(L, R)
{
  constexpr auto pretty_L = k3::tok3n::pretty(L{}).view();
  constexpr auto pretty_R = k3::tok3n::pretty(R{}).view();

  if constexpr (std::same_as<typename L::value_type, typename R::value_type>)
  {
    ASSERT((can_sequence<L, R>),
      std::string("`L >> R` did not compile\nL = ")
        + pretty_L + "\nR = " + pretty_R);
  }
  else
  {
    ASSERT((!can_sequence<L, R>),
      std::string("`L >> R` compiled but it should not\nL = ")
        + pretty_L + "\nR = " + pretty_R);
  }
}
```

Conclusion

Summary

- `tok3n` is a parser combinator library for spans of anything
- I need to test it at compile-time
- CT testing is hard, especially custom error messages
- Test frameworks Snitch and UT
- CT test failures don't need to fail the compile!
- If we want `static_assert` test cases, it'll be better in C++26

Takeaways

- Try out new libraries, you might learn something
- Dive deep and reassess your assumptions
- Javascript is actually pretty cool

Thank you!

Braden Ganetsky

braden@ganets.ky
GitHub @k3DW