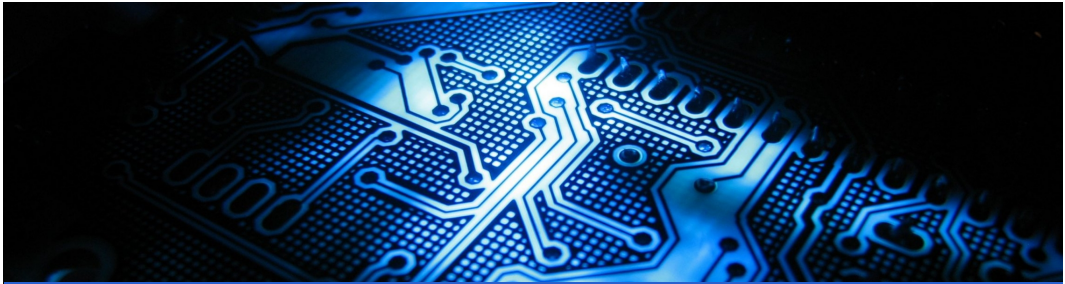

lexy

A parser DSL library

Jonathan Müller — @foonathan — CC BY 4.0

Roadmap

- Quick introduction to lexy
- How lexy works:
  - How to represent input?
  - How to define the grammar?
  - How to produce values?

What is lexy?

C++ Parser DSL

- syntax sugar for a recursive descent parser
- full control over branching decisions, backtracking
- bring your own data structure
- Unicode support, automatic error recovery, parse tree generation

lexy 101

How to use lexy in three steps

1. Define the grammar.
2. Create an input.
3. Call a parse action.

lexy 101

How to use lexy in three steps

1. Define the grammar.
2. Create an input.
3. Call a parse action.

Example: parse HTML color (`#FF00FF`) as `Color`.

```
struct Color
{
    std::uint8_t r, g, b;
};
```

1. Define the grammar

Grammar consists of *productions*, which are `structs`.

1. Define the grammar

Grammar consists of *productions*, which are `structs`.

```
struct color
{
    static constexpr auto rule
      = dsl::hash_sign + dsl::times<3>(dsl::p<channel>);
    static constexpr auto value = lexy::construct<Color>;
};
```

1. Define the grammar

Grammar consists of *productions*, which are `structs`.

```
struct color
{
    static constexpr auto rule
      = dsl::hash_sign + dsl::times<3>(dsl::p<channel>);
    static constexpr auto value = lexy::construct<Color>;
};
```

```
struct channel
{
    static constexpr auto rule
      = dsl::integer<std::uint8_t>(dsl::n_digits<2, dsl::hex>);
    static constexpr auto value = lexy::forward<std::uint8_t>;
};
```

2. Create an input

String:

```
auto input = lexy::zstring_input("#FF00FF");
```

2. Create an input

String:

```
auto input = lexy::zstring_input("#FF00FF");
```

File:

```
auto file = lexy::read_file<lexy::utf8_encoding>(path);
if (!file) { … }
auto input = file.buffer();
```

2. Create an input

String:

```
auto input = lexy::zstring_input("#FF00FF");
```

File:

```
auto file = lexy::read_file<lexy::utf8_encoding>(path);
if (!file) { … }
auto input = file.buffer();
```

Iterator range:

```
auto input = lexy::range_input<lexy::ascii_encoding>(begin, end);
```

2. Create an input

String:

```
auto input = lexy::zstring_input("#FF00FF");
```

File:

```
auto file = lexy::read_file<lexy::utf8_encoding>(path);
if (!file) { … }
auto input = file.buffer();
```

Iterator range:

```
auto input = lexy::range_input<lexy::ascii_encoding>(begin, end);
```

Command-line arguments:

```
auto input = lexy::argv_input(argc, argv);
```

3. Call a parse action

DEMO

Parse the grammar into a data structure:

```
auto result = lexy::parse<grammar::color>(input, lexy_ext::report_error);
if (result.has_value())
{
    Color color = result.value();
    …
}
```

3. Call a parse action

DEMO

Parse the grammar as a lossless parse tree:

```
lexy::parse_tree_for<decltype(input)> tree;
auto result = lexy::parse_as_tree<grammar::color>(tree, input,
                                                  lexy_ext::report_error);
lexy::visualize(stdout, tree);
```

3. Call a parse action

DEMO

Parse the grammar as a lossless parse tree:

```
lexy::parse_tree_for<decltype(input)> tree;
auto result = lexy::parse_as_tree<grammar::color>(tree, input,
                                                  lexy_ext::report_error);
lexy::visualize(stdout, tree);
```

Debug parsing algorithm:

```
lexy::trace<grammar::color>(stdout, input);
```

lexy is not declarative

Regex: `a*a`

`a`, `aa`, `aaa`, `aaaa`, ...

lexy is not declarative

Regex: `a*a`

`a`, `aa`, `aaa`, `aaaa`, ...

lexy

```
dsl::while_(dsl::lit_c<'a'>) + dsl::lit_c<'a'>;
```

Does not match anything!

lexy is not declarative

Regex: `a*a`

a, aa, aaa, aaaa, ...

lexy

```
dsl::while_(dsl::lit_c<'a'>) + dsl::lit_c<'a'>;
```

Does not match anything!

```
while (match('a'))
    consume('a');

match_and_consume('a');
```

lexy is not declarative

Regex: `a|ab`

`a`, `ab`

lexy is not declarative

Regex: `a|ab`

`a, ab`

lexy

```
dsl::lit_c<'a'> | dsl::lit<"ab">
```

`a`

lexy is not declarative

Regex: `a|ab`

`a, ab`

lexy

```
dsl::lit_c<'a'> | dsl::lit<"ab">
```

`a`

```
if (match('a'))
    consume('a');
else if (match("ab"))
    consume("ab");
else
    error();
```

lexy has explicit backtracking

Decisions are made using branch conditions

- tokens: `dsl::lit`, `dsl::ascii::*`, ... (LL(1) lookahead)
- atomic conditions: `dsl::peek(rule)`, `dsl::else_`, ...
- branch rules: `condition >> then`

lexy has explicit backtracking

DEMO

Example: parse HTML color (`#FF00FF`) or function call (`rgb(a, b, c)`).

```
// #FF00FF
auto hex_color = dsl::hash_sign >> dsl::times<3>(dsl::p<channel_hex>);

// rgb(255, 0, 255)
auto three_channel_dec = dsl::times<3>(dsl::p<channel_dec>,
                                       dsl::sep(dsl::comma));
auto rgb_function
    = LEXY_LIT("rgb") >> dsl::parenthesized(three_channel_dec);

return hex_color | rgb_function;
```

lexy.foonathan.net

Other features

- mix DSL with hand-written parser with `dsl::scan`
- query Unicode character properties (`constexpr`)
- byte input: big/little endian integers, specific bits, TLV
- keywords and identifiers, soon™ operator precedence

Version 1.0 comes January 2022!

How to represent input?

How to represent input?

Design Goal

Support arbitrary input sources (files, strings, ...).

How to represent input?

Design Goal

Support arbitrary input sources (files, strings, ...).

Iterator range `[begin, end)` as input!

Iterator range as input

```
template <typename Iterator>
bool parse_a(Iterator& cur, Iterator end)
{
    if (cur == end || *cur != 'a')
        return false;

    ++cur;
    return true;
}
```

```
template <typename Iterator>
bool parse_grammar(Iterator& cur, Iterator end)
{
    …
}
```

Iterator range as input

Use a file as input:

```
std::string str = read_file(…);
auto cur = str.begin();
auto result = parse_grammar(str.begin(), str.end());
```

Iterator range as input

Use a file as input:

```
std::string str = read_file(…);
auto cur = str.begin();
auto result = parse_grammar(str.begin(), str.end());
```

Use a string literal as input:

```
auto input = "Hello World!";
auto cur = input;
auto result = parse_grammar(cur, input + std::strlen(input));
```

Sentinels

Iterator:

```
cur == end
```

Null-terminated string:

```
*cur == '\0'
```

Sentinels

Iterator:

```
cur == end
```

Null-terminated string:

```
*cur == '\0'
```

Creative operator overloading:

```
struct zstring_sentinel
{
    friend bool operator==(const char* iterator, zstring_sentinel)
    {
        return *iterator == '\0';
    }

    // reversed arguments, operator!= not needed in C++20
};
```

Sentinels

```
template <typename Iterator, typename Sentinel>
bool parse_a(Iterator& cur, Sentinel end)
{
    // unchanged!
}
```

```
auto input = "Hello World!";

auto cur = input;
auto parse = parse_grammar(cur, zstring_sentinel{});
```

Sentinels

```cpp
template <typename Iterator, typename Sentinel>
bool parse_a(Iterator& cur, Sentinel end)
{
    // unchanged!
}
```

```cpp
auto input = "Hello World!";

auto cur = input;
auto parse = parse_grammar(cur, zstring_sentinel{});
```

Part of C++20: www.foonathan.net/2020/03/iterator-sentinel/.

Sentinels

```
if (cur == end || *cur != 'a')
    return false;
```

Sentinels

```
if (cur == end || *cur != 'a')
    return false;
```

really means

```
if (*cur == '\0' || *cur != 'a')
    return false;
```

Sentinels

```
if (cur == end || *cur != 'a')
    return false;
```

really means

```
if (*cur == '\0' || *cur != 'a')
    return false;
```

so why not?

```
if (*cur != 'a')
    return false;
```

REPL: Goal

DEMO

Design Goal

Support a REPL

REPL: Goal

DEMO

Design Goal

Support a REPL

```
> echo Hello
Hello
> set name \
. "Jonathan Müller"
> echo $name
Jonathan Müller
```

REPL: Goal

Write a `[begin, end)` that dynamically requests input:

- Initially prints > and asks for a line of input.
- When end of input is reached, but grammar not done, re-prompt by printing `.` and asking for more input.
- EOF reached when re-prompting results in an empty line.

REPL: Goal

Write a `[begin, end)` that dynamically requests input:

- Initially prints `>` and asks for a line of input.
- When end of input is reached, but grammar not done, re-prompt by printing `.` and asking for more input.
- EOF reached when re-prompting results in an empty line.

`repl_iterator`, `repl_sentinel`:

- `operator*`
- `operator++`
- `operator==`

REPL: Take 1

```
struct repl_iterator
{
    std::string line;
    std::string::iterator pos;

    repl_iterator()
    {
        std::cout << "> ";
        std::getline(std::cin, line);
        pos = line.begin();
    }
};
```

REPL: Take 1

```
struct repl_iterator
{
    char operator*() const { return *pos; }

    repl_iterator& operator++()
    {
        if (++pos == line.end())
        {
            std::cout << ". ";
            std::getline(std::cin, line);
            pos = line.begin();
        }
        return *this;
    }
};
```

REPL: Take 1

```
struct repl_sentinel
{
    friend bool operator==(const repl_iterator& iterator, repl_sentinel)
    {
        return iterator.line.empty();
    }
};
```

REPL: Take 1

```cpp
template <typename Iterator, typename Sentinel>
bool parse_abc(Iterator& cur, Sentinel end)
{
    if (cur == end || *cur != 'a')
        return false;
    ++cur;
    if (cur == end || *cur != 'b')
        return false;
    ++cur;
    if (cur == end || *cur != 'c')
        return false;
    ++cur;
    return true;
}
```

REPL: Take 1

```
auto cur = repl_iterator{};
auto result = parse_abc(cur, repl_sentinel{});
if (result)
    std::cout << "yes!\n";
else
    std::cout << "no :(\n";
```

REPL: Take 1

```
auto cur = repl_iterator{};
auto result = parse_abc(cur, repl_sentinel{});
if (result)
    std::cout << "yes!\n";
else
    std::cout << "no :(\n";
```

```
> xyz
no :(
```

REPL: Take 1

```
auto cur = repl_iterator{};
auto result = parse_abc(cur, repl_sentinel{});
if (result)
    std::cout << "yes!\n";
else
    std::cout << "no :(\n";
```

```
> xyz
no :(

> abc
.
```

REPL: Take 1

```cpp
template <typename Iterator, typename Sentinel>
bool parse_abc(Iterator& cur, Sentinel end)
{
    if (cur == end || *cur != 'a')
        return false;
    ++cur;                              // consume a
    if (cur == end || *cur != 'b')
        return false;
    ++cur;                              // consume b
    if (cur == end || *cur != 'c')
        return false;
    ++cur;                              // consume c + request line!
    return true;
}
```

REPL: Take 1

```cpp
template <typename Iterator, typename Sentinel>
bool parse_abc(Iterator& cur, Sentinel end)
{
    if (cur == end || *cur != 'a')
        return false;
    ++cur;                              // consume a
    if (cur == end || *cur != 'b')
        return false;
    ++cur;                              // consume b
    if (cur == end || *cur != 'c')
        return false;
    ++cur;                              // consume c + request line!
    return true;
}
```

We must not re-prompt in `operator++`!

REPL: Take 2

```
repl_iterator& operator++()
{
    ++pos;
    return *this;
}
char operator*() const
{
    if (pos == line.end())
    {
        std::cout << ". ";
        std::getline(std::cin, line);
        pos = line.begin();
    }
    return *pos; // what if now line.empty()?
}
```

REPL: Take 2

Infinite loop:

```
template <typename Iterator, typename Sentinel>
bool parse_rest(Iterator& cur, Sentinel end)
{
    while (cur != end)
        ++cur;
    return true;
}
```

REPL: Take 2

Infinite loop:

```
template <typename Iterator, typename Sentinel>
bool parse_rest(Iterator& cur, Sentinel end)
{
    while (cur != end)
        ++cur;
    return true;
}
```

We cannot re-prompt in `operator*`!

REPL: Take 3

```
bool operator==(const repl_iterator& iterator, repl_sentinel)
{
    if (auto iter = const_cast<repl_iterator&>(iterator);
        iter.pos == iter.line.end())
    {
        std::cout << ". ";
        std::getline(std::cin, iter.line);
        iter.pos = iter.line.begin();
    }
    return iterator.line.empty();
}
```

REPL: Take 3

```
bool operator==(const repl_iterator& iterator, repl_sentinel)
{
    if (auto iter = const_cast<repl_iterator&>(iterator);
        iter.pos == iter.line.end())
    {
        std::cout << ". ";
        std::getline(std::cin, iter.line);
        iter.pos = iter.line.begin();
    }
    return iterator.line.empty();
}
```

This works!

REPL: Re-visiting the iterator model

We need an `operator*` that can fail:

```
std::optional<char> operator*() const
{
    if (pos == line.end()) { … }

    return pos == line.end() ? std::nullopt : *pos;
}
```

```
bool operator==(const repl_iterator& iterator, repl_sentinel)
{
    return !iterator->has_value();
}
```

REPL: Re-visiting the iterator model

We need an `operator*` that can fail:

```
std::optional<char> operator*() const
{
    if (pos == line.end()) { … }

    return pos == line.end() ? std::nullopt : *pos;
}
```

```
bool operator==(const repl_iterator& iterator, repl_sentinel)
{
    return !iterator->has_value();
}
```

Why even have `operator==` now? Or iterator ranges?

The Reader[1]

```
struct Reader
{
    std::optional<char> peek(); // operator* + operator==

    void bump();                // operator++
};
```

[1] 50% true.

The Reader

```
template <typename Reader>
bool parse_a(Reader& reader)
{
    if (reader.peek() != 'a')
        return false;

    reader.bump();
    return true;
}
```

The Reader

```
template <typename Iterator, typename Sentinel = Iterator>
struct range_reader
{
    Iterator cur;
    Sentinel end;

    std::optional<char> peek()
    {
        return cur == end ? std::nullopt : *cur;
    }

    void bump() { ++cur; }
};
```

Re-visiting null-terminated strings

```
struct zstring_reader
{
    const char* cur;

    std::optional<char> peek()
    {
        return *cur ? *cur : std::nullopt;
    }

    void bump()
    {
        ++cur;
    }
};
```

The Reader[2]

```
struct Reader
{
    static constexpr char_or_empty eof;

    char_or_empty peek();

    void bump();
};
```

[2] 60% true.

Re-visiting null-terminated strings

```
struct zstring_reader
{
    const char* cur;

    static constexpr auto eof = '\0';

    char peek()
    {
        return *cur; // no branch!
    }

    void bump()
    {
        ++cur;
    }
};
```

How to represent input?

The `Reader`

- `peek()` + `bump()` method
- can wrap any iterator range
- supports REPL
- optimize input with sentinel values (zero-terminated strings, UTF-8, ...)

How to represent input?

The `Reader`

- `peek()` + `bump()` method
- can wrap any iterator range
- supports REPL
- optimize input with sentinel values (zero-terminated strings, UTF-8, …)

Barry Revzin's talk: Iterators and Ranges – Comparing C++ to D to Rust

How to define the grammar?

How to define the grammar?

Design Goals

- Essentially syntax sugar for a hand-written parser
- Full control over backtracking
- Build grammars out of small rules

How to define the grammar?

Design Goals

- Essentially syntax sugar for a hand-written parser
- Full control over backtracking
- Build grammars out of small rules

Combinatory parsing.

Combinatory parsing

Parser: `Reader -> (bool, Reader)`

```
template <typename Reader>
bool parse(Reader& reader);
```

Combinatory parsing

Parser: `Reader -> (bool, Reader)`

```
template <typename Reader>
bool parse(Reader& reader);
```

```
template <typename Reader>
bool parse_a(Reader& reader)
{
    if (reader.peek() != 'a')
        return false;

    reader.bump();
    return true;
}
```

Combinatory parsing: Parametrization

```
template <typename Reader>
bool parse_lit_c(Reader& reader, char c)
{
    if (reader.peek() != c)
        return false;

    reader.bump();
    return true;
}
```

Combinatory parsing: Parametrization

```cpp
template <typename Reader>
bool parse_lit_c(Reader& reader, char c)
{
    if (reader.peek() != c)
        return false;

    reader.bump();
    return true;
}
```

Problem: Does not compose nicely!

```cpp
auto result = parse_lit_c(reader, 'a');
```

Combinatory parsing: Higher-order functions

```
auto lit_c(char c)
{
    return [=](auto& reader) {
        if (reader.peek() != c)
            return false;

        reader.bump();
        return true;
    };
}
```

Combinatory parsing: Higher-order functions

```
auto lit_c(char c)
{
    return [=](auto& reader) {
        if (reader.peek() != c)
            return false;

        reader.bump();
        return true;
    };
}
```

```
auto rule = lit_c('a');
…
auto result = rule(reader);
```

Combinatory parsing

Parser combinator: `(Parser_0, ... Parser_n) -> Parser`

Combinatory parsing

Parser combinator: `(Parser_0, ... Parser_n) -> Parser`

```
template <typename P1, typename P2>
auto sequence(P1 p1, P2 p2)
{
    return [=](auto& reader) {
        if (!p1(reader))
            return false;

        return p2(reader);
    };
}
```

```
auto rule = sequence(lit_c('a'), sequence(lit_c('b'), lit_c('c')));
…
auto result = rule(reader);
```

Fold expressions are awesome, part 1

```
template <typename ... Ps>
auto sequence(Ps... ps)
{
    return [=](auto& reader) {
        return (ps(reader) && ...);
    };
}
```

```
auto rule = sequence(lit_c('a'), lit_c('b'), lit_c('c'));
```

Combinatory parsing: Making decisions

```
template <typename P1, typename P2>
auto branch(P1 condition, P2 then)
{
    return [=](auto& reader) {
        auto marker = reader;
        if (condition(reader))
            return then(reader);

        reader = marker; // backtrack
        return false;
    };
}
```

Combinatory parsing: Making decisions

```
template <typename P1, typename P2>
auto choice(P1 p1, P2 p2) // assuming P1 is a branch
{
    return [=](auto& reader) {
        if (p1(reader))
            return true;

        return p2(reader);
    };
}
```

Combinatory parsing: Making decisions

```
template <typename P1, typename P2>
auto choice(P1 p1, P2 p2) // assuming P1 is a branch
{
    return [=](auto& reader) {
        if (p1(reader))
            return true;

        return p2(reader);
    };
}
```

```
auto rule = choice(branch(lit_c('a'), lit_c('b')), lit_c('b'));
```

Fold expressions are awesome, part 2

```
template <typename ... Ps>
auto sequence(Ps... ps)
{
    return [=](auto& reader) {
        return (ps(reader) && ...);
    };
}
```

Fold expressions are awesome, part 2

```
template <typename ... Ps>
auto sequence(Ps... ps)
{
    return [=](auto& reader) {
        return (ps(reader) && ...);
    };
}
```

```
template <typename ... Ps>
auto choice(Ps... ps)
{
    return [=](auto& reader) {
        return (ps(reader) || ...);
    };
}
```

How to define the grammar?

Problems with higher-order functions

- Verbose and ugly syntax
- Overhead due to stateful lambda

Type-based combinatory parsing

```
struct lit_c_parser
{
    char c;

    template <typename Reader>
    bool operator()(Reader& reader) const { … }
};

auto lit_c(char c)
{
    return lit_c_parser{c};
}
```

Type-based combinatory parsing

```
template <char C>
struct lit_c_parser
{
    template <typename Reader>
    static bool parse(Reader& reader) { … }
};
```

Type-based combinatory parsing

```
template <typename .... Ps>
struct sequence_parser
{
    std::tuple<Ps...> ps;

    template <typename Reader>
    bool operator()(Reader& reader) const { … }
};

template <typename ... Ps>
auto sequence(Ps... ps)
{
    return sequence_parser{ps...};
}
```

Type-based combinatory parsing

```
template <typename ... Ps>
struct sequence_parser
{
    template <typename Reader>
    static bool parse(Reader& reader) const
    {
      return (Ps::parse(reader) && ...);
    }
};
```

Type-based combinatory parsing

Before:

```
auto rule = sequence(lit_c('a'), lit_c('b'), lit_c('c'));
```

After:

```
using rule = sequence_parser<lit_c_parser<'a'>, lit_c_parser<'b'>,
                             lit_c_parser<'c'>>;
```

Type-based combinatory parsing

Before:

```
auto rule = sequence(lit_c('a'), lit_c('b'), lit_c('c'));
```

After:

```
using rule = sequence_parser<lit_c_parser<'a'>, lit_c_parser<'b'>,
                             lit_c_parser<'c'>>;
```

PEGTL: github.com/taocpp/PEGTL

Adding a DSL on-top

```
template <char C>
struct lit_c_parser { … };

template <char C>
constexpr auto lit_c = lit_c_parser<C>{};
```

```
template <typename ... Ps>
struct sequence_parser { … };

template <typename ... Ps>
constexpr auto sequence(Ps... /* ps */)
{
    return sequence_parser<Ps...>{};
}
```

Adding a DSL on-top

```
template <typename P1, typename P2>
constexpr auto operator+(P1, P2)
{
    return sequence_parser<P1, P2>{};
}

template <typename P1, typename P2>
constexpr auto operator|(P1, P2)
{
    return choice_parser<P1, P2>{};
}
```

How to define the grammar?

The DSL

- Parser: type with a `parse()` function
- Parser combinator: combines multiple parsers
- Grammar expressed in the type system:
  DSL types are empty, value doesn't matter

How to produce values?

How to produce values?

```cpp
template <typename IntT, typename Digits>
struct integer_parser
{
    template <typename Reader>
    static bool parse(Reader& reader)
    {
        auto begin = reader.position();
        if (!Digits::parse(reader))
            return false;
        auto end = reader.position();

        IntT result;
        std::from_chars(begin, end, result);
        // ... now what?
    }
};
```

Parser returns value

Parser returns value

New Parser: `Reader -> (T?, Reader)`

```
template <typename Reader>
std::optional<T> parse(Reader& reader);
```

Parser returns value

New Parser: `Reader -> (T?, Reader)`

```
template <typename Reader>
std::optional<T> parse(Reader& reader);
```

```
template <typename IntT, typename Digits>
struct integer_parser
{
    template <typename Reader>
    static std::optional<IntT> parse(Reader& reader) { … }
};
```

Parser returns value

```
integer IntT
```

Parser returns value

```
integer IntT
  lit_c ~~void~~ struct Void {};
```

Parser returns value

```
 integer IntT
   lit_c ~~void~~ struct Void {};
sequence std::tuple
```

Parser returns value

```
 integer IntT
   lit_c ~~void~~ struct Void {};
sequence std::tuple
  choice std::variant
```

Parser returns value

```
 integer IntT
   lit_c ~~void~~ struct Void {};
sequence std::tuple
  choice std::variant
    list std::vector
```

Parser returns value

```
auto rule = lit_c<'a'>
          + (lit_c<'a'> >> lit_c<'b'> + integer)
              | (lit_c<'b'> >> lit_c<'c'>)
          + list(lit_c<'x'>)
          + integer;
```

Parser returns value

```
auto rule = lit_c<'a'>
          + (lit_c<'a'> >> lit_c<'b'> + integer)
              | (lit_c<'b'> >> lit_c<'c'>)
          + list(lit_c<'x'>)
          + integer;
```

```
std::tuple<Void,
      std::variant<std::tuple<Void, Void, int>, std::tuple<Void, Void>>,
      std::vector<Void>,
      int>
```

Parser returns value

Simplification rules:

- `std::tuple<Void, Void>` → `Void`
- `std::tuple<Void, T>` → `T`

Parser returns value

Simplification rules:

- `std::tuple<Void, Void>` → `Void`
- `std::tuple<Void, T>` → `T`
- `std::variant<T, Void>` → `std::optional<T>`
- `std::variant<T, T>` → `T`

Parser returns value

Simplification rules:

- `std::tuple<Void, Void>` → `Void`
- `std::tuple<Void, T>` → `T`
- `std::variant<T, Void>` → `std::optional<T>`
- `std::variant<T, T>` → `T`
- `std::optional<Void>` → `bool`
- `std::vector<Void>` → `std::size_t`

Parser returns value

Design Goal

Bring your own data structure.

Parser returns value

Design Goal

Bring your own data structure.

Generalize `tuple` and `variant` to user-defined types.

Parser returns value

Design Goal

Bring your own data structure.

Generalize `tuple` and `variant` to user-defined types.

Boost.Spirit: boost-spirit.com

Parser returns value

Problems

- Unwieldy return types without simplification rules
- Simplification rules add complexity
- Supporting user-defined types add complexity

Interlude: Coroutines

Interlude: Coroutines

```
Task<int> compute_expensive(int arg)
{
    auto result = co_await implementation(arg);
    co_await process(result);
    co_return result;
}
```

Interlude: Coroutines

```
Task<int> compute_expensive(int arg)
{
    auto result = co_await implementation(arg);
    co_await process(result);
    co_return result;
}
```

```
Task<int> compute_sum_bad()
{
    auto a = co_await compute_expensive(0);
    auto b = co_await compute_expensive(1);
    auto c = co_await compute_expensive(2);
    return a + b + c;
}
```

Interlude: Coroutines

```
Task<int> compute_sum_good()
{
    auto [a, b, c]
      = co_await when_all(compute_expensive(0),
                          compute_expensive(1),
                          compute_expensive(2));
    co_return a + b + c;
}
```

Interlude: Coroutines

```
Task<int> compute_sum_good()
{
    auto [a, b, c]
      = co_await when_all(compute_expensive(0),
                          compute_expensive(1),
                          compute_expensive(2));
    co_return a + b + c;
}
```

```
template <typename ... Tasks>
auto when_all(Task... tasks)
  -> Task<std::tuple<…>>
{
    …
}
```

Interlude: Coroutines

```
Task<int> compute_first()
{
    auto result
      = co_await when_any(compute_expensive(0),
                          compute_expensive(1),
                          compute_expensive(2));
    co_return result;
}
```

Interlude: Coroutines

```
Task<int> compute_first()
{
    auto result
      = co_await when_any(compute_expensive(0),
                          compute_expensive(1),
                          compute_expensive(2));
    co_return result;
}
```

```
template <typename ... Tasks>
auto when_any(Task... tasks)
  -> Task<std::variant<…>>
{
    …
}
```

Interlude: Coroutines

Sender/Receiver model

- `Sender` describes work that should be done
- `Receiver` is a generalized callback that receives the result
- executing a connected `Sender`/`Receiver` pair computes the result and passes it to the callback

Eric Niebler's talk: A Tour of C++ executors

Interlude: Coroutines

The receiver concept[3]

```
struct Receiver
{
    template <typename ... Args>
    void set_value(Args&&... args);

    template <typename Error>
    void set_error(Error&& error);

    void set_done();
};
```

[3] simplified

Parser invokes callback

Parser invokes callback

```
lit_c callback()
```

Parser invokes callback

```
  lit_c callback()
integer callback(IntT)
```

Parser invokes callback

```
   lit_c callback()
 integer callback(IntT)
sequence callback(Args...)
```

Parser invokes callback

```
   lit_c callback()
 integer callback(IntT)
sequence callback(Args...)
  choice callback(Args1...) or callback(Args2...) or ...
```

Parser invokes callback

```
template <char C>
struct lit_c_parser
{
    template <typename Callback, typename Reader>
    static bool parse(Callback& cb, Reader& reader)
    {
        if (reader.peek() != C)
            return false;
        reader.bump();
        cb();
        return true;
    }
};
```

Parser invokes callback

```
template <typename IntT, typename Digits>
struct integer_parser
{
    template <typename Callback, typename Reader>
    static bool parse(Callback& cb, Reader& reader)
    {
        …

        IntT result;
        std::from_chars(begin, end, result);
        cb(result);
        return true;
    }
};
```

Parser invokes callback

```
template <typename ... Ps>
struct sequence_parser
{
    template <typename Callback, typename Reader>
    static bool parse(Callback& cb, Reader& reader)
    {
        return (Ps::parse(cb, reader) && ...);
    }
};
```

Parser invokes callback

```
template <typename ... Ps>
struct sequence_parser
{
    template <typename Callback, typename Reader>
    static bool parse(Callback& cb, Reader& reader)
    {
        return (Ps::parse(cb, reader) && ...);
    }
};
```

This invokes the callback `sizeof...(Ps)` times!

Parser invokes callback

```
sequence(lit_c<'a'>, lit_c<'b'>, lit_c<'c'>)
```

Parser invokes callback

```
sequence(lit_c<'a'>, lit_c<'b'>, lit_c<'c'>)
```

```
sequence:
  - lit_c<'a'>:
    - callback()
  - lit_c<'b'>:
    - callback()
  - lit_c<'c'>:
    - callback()
```

Parser invokes callback

```
sequence(lit_c<'a'>, lit_c<'b'>, lit_c<'c'>)
```

```
sequence:
  - lit_c<'a'>:
    - callback()
  - lit_c<'b'>:
    - callback()
  - lit_c<'c'>:
    - callback()
```

```
sequence:
  - lit_c<'a'>:
    - lit_c<'b'>:
      - lit_c<'c'>:
        - callback()
```

Parser has continuation

Parser has continuation

- Rule: DSL object, e.g. `lit_c<'a'>`
- Rule has a `template <typename NextParser> struct p` member
- Parser: has a `parse()` function that parses and forwards to `NextParser`

Parser has continuation

- Rule: DSL object, e.g. `lit_c<'a'>`
- Rule has a `template <typename NextParser> struct p` member
- Parser: has a `parse()` function that parses and forwards to `NextParser`

```
template <typename Rule, typename NextParser>
using parser_for = typename Rule::template p<NextParser>;
```

Parser has continuation

```
struct Rule
{
    template <typename NextParser>
    struct p
    {
        template <typename Callback, typename Reader, typename ... Args>
        static bool parse(Callback& cb, Reader& reader,
                          Args... prev_args)
        {
            if (!matches(reader))
                return false;
            return NextParser::parse(cb, reader, prev_args..., value);
        }
    };
};
```

Parser has continuation

```
template <char C>
struct lit_c_rule
{
    template <typename NextParser>
    struct p
    {
        template <typename Callback, typename Reader, typename ... Args>
        static bool parse(Callback& cb, Reader& reader,
                          Args... prev_args)
        {
            if (reader.peek() != C) return false;
            reader.bump();
            return NextParser::parse(cb, reader, prev_args...);
        }
    };
};
```

Parser has continuation

```
template <typename T, typename Digits>
struct integer_rule
{
    template <typename NextParser>
    struct p
    {
        template <typename Callback, typename Reader, typename ... Args>
        static bool parse(Callback& cb, Reader& reader,
                          Args... prev_args)
        {
            …
            return NextParser::parse(cb, reader,
                                     prev_args..., result);
        }
    };
};
```

Parser has continuation

```
struct final_parser
{
    template <typename Callback, typename Reader, typename ... Args>
    static bool parse(Callback& cb, Reader& reader, Args... args)
    {
        cb(args...);
        return true;
    }
};
```

```
template <typename Rule, typename Callback, typename Reader>
bool parse_rule(Callback& cb, Reader& reader)
{
    return parser_for<Rule, final_parser>::parse(cb, reader);
}
```

Parser has continuation

```
template <typename R1, typename R2>
struct sequence_rule
{
    template <typename NextParser>
    struct p
    {
        template <typename Callback, typename Reader, typename ... Args>
        static bool parse(Callback& cb, Reader& reader,
                          Args... prev_args)
        {
            using tail = parser_for<R2, NextParser>;
            using head = parser_for<R1, tail>;
            return head::parse(cb, reader, prev_args...);
        }
    };
};
```

Parser has continuation

```
template <typename R1, typename R2>
struct sequence_rule
{
    template <typename NextParser>
    using p = parser_for<R1, parser_for<R2, NextParser>>;
};
```

Sequence is completely free!

Parser has continuation

```
template <typename ... Rs>
struct choice_rule
{
    template <typename NextParser>
    struct p
    {
        template <typename Callback, typename Reader, typename ... Args>
        static bool parse(Callback& cb, Reader& reader,
                          Args... prev_args)
        {
            return (parser_for<Rs, NextParser>::parse(cb, reader, prev_args...)
                    || ...);
        }
    };
};
```

Parser has continuation

```
choice(branch(lit_c<'a'>, lit_c<'b'>), branch(lit_c<'b'>, integer))
```

Parser has continuation

```
choice(branch(lit_c<'a'>, lit_c<'b'>), branch(lit_c<'b'>, integer))
```

```
choice:
  - branch:
    - lit_c<'a'>:
      - lit_c<'b'>:
        - callback()
```

Parser has continuation

```
choice(branch(lit_c<'a'>, lit_c<'b'>), branch(lit_c<'b'>, integer))
```

```
choice:
  - branch:
    - lit_c<'a'>:
      - lit_c<'b'>:
        - callback()
```

or

```
choice:
  - branch:
    - lit_c<'a'>:
      - integer:
        - callback()
```

What about list?

List produces a dynamic amount of values.

What about list?

List produces a dynamic amount of values.

```
// Parse a list of Item.
template <typename Callback, typename Reader, typename ... Args>
static bool parse(Callback& cb, Reader& reader,
                  Args... prev_args)
{
    auto sink = cb.sink();

    while (parser_for<Item, final_parser>::parse(sink, reader))
    {}

    return NextParser::parse(cb, reader, prev_args..., sink.finish());
}
```

How to produce values?

Callbacks

- multiple arguments instead of tuple, overload sets instead of variant
- parsers connected with continuation
- all values accumulated in the parameters
- sequence is free!

Conclusion

Takeaways

- Look for simpler/different models if an existing one doesn't fit well
- Put compile-time information into the type system
- Callbacks are in some ways superior to return values
- Fold expressions are awesome

lexy: lexy.foonathan.net

Twitter: @foonathan

Support me: jonathanmueller.dev/support-me/