

# lexy's text and Unicode challenges

Jonathan Müller — @foonathan — CC BY 4.0

Parse combinator library: [github.com/foonathan/lexy](https://github.com/foonathan/lexy)

```
struct ipv4_address
{
    static constexpr auto rule = []{
        auto octet = dsl::integer<std::uint8_t>(dsl::digits<>);

        return dsl::times<4>(octet, dsl::sep(dsl::period)) + dsl::eof;
    }();
};

auto input  = lexy::zstring_input("192.168.1.1");
auto result = lexy::parse<ipv4_address>(input, /* error callback */);
```

# Inputs and Encodings

# The Input concept

```
class Input
{
public:
    Reader reader() const &;
};
```

- `lexy::range_input: [begin, end)`
- `lexy::string_input: string literal, std::string_view, std::string`
- `lexy::buffer: owning container`

# Potential Reader concept

```
class Reader
{
public:
    bool eof() const { return /* eof? */; }

    auto peek() const { return /* current character */; }

    void bump() { /* advance to the next position */ }
};
```

```
template <typename Reader>
bool parse_42(Reader& reader)
{
    if (reader.eof() || reader.peek() != '4')
        return false;
    reader.bump();

    if (reader.eof() || reader.peek() != '2')
        return false;
    reader.bump();

    return true;
}
```

```
template <typename Reader>
bool parse_42(Reader& reader)
{
    if (reader.peek() != '4')
        return false;
    reader.bump();

    if (reader.peek() != '2')
        return false;
    reader.bump();

    return true;
}
```

lexy::buffer: sentinel EOF value at the end.

# Actual Reader concept

```
class Reader
{
public:
    ??? peek() const
    {
        return /* current character or EOF */;
    }

    void bump()
    {
        /* advance to the next position */
    }
};
```



# Encoding concept

Subset of `std::char_traits`:

```
class Encoding
{
public:
    using char_type = ...;
    using int_type  = ...;

    static constexpr int_type eof();

    static int_type to_int_type(char_type c);
};
```

# Why not `std::char_traits`?

- Too many unnecessary methods for my use case
- Not enough methods for my use case
- Only one encoding per character
- (I didn't want to drag entire `<string>` header into everything)

# Why not `std::char_traits`?

- Too many unnecessary methods for my use case
- Not enough methods for my use case
- Only one encoding per character
- (I didn't want to drag entire `<string>` header into everything)
- `int_type` too big

# Encodings

Encoding	char_type	int_type	EOF
default	char	int	-1
raw	unsigned char	int	-1
ASCII	char	char	0xFF
UTF-8	char8_t	char8_t	0xFF
UTF-16	char16_t	std::int_least32_t	-1
UTF-32	char32_t	char32_t	0xFFFF'FFFF

Encoding	char_type	int_type	EOF
default	char	int	-1
raw	unsigned char	int	-1
ASCII	char	char	0xFF
UTF-8	char8_t	char8_t	0xFF
UTF-16	char16_t	std::int_least32_t	-1
UTF-32	char32_t	char32_t	0xFFFF'FFFF

Potentially in the future: “tokenized” encoding.

# Encoded Inputs

```
template <typename Encoding>
class string_input;

template <typename Encoding, typename CharT>
string_input<Encoding> zstring_input(const CharT* str);

...

zstring_input<ascii_encoding>("ASCII");
zstring_input<utf16_encoding>(u"UTF-16");

unsigned char raw[] = {...};
string_input<raw_encoding> input(std::begin(raw), std::end(raw));
```

# Deducing Encodings

```
zstring_input("default encoding");  
zstring_input(u8"UTF-8 encoding");  
zstring_input(u"UTF-16 encoding");  
zstring_input(U"UTF-32 encoding");  
  
unsigned char raw[] = {...};  
string_input input(std::begin(raw), std::end(raw));
```

## Secondary char\_type

Encoding	char_type	secondary char type
default	char	none
raw	unsigned char	char (and std::byte)
ASCII	char	none
UTF-8	char8_t	char
UTF-16	char16_t	wchar_t (if applicable)
UTF-32	char32_t	wchar_t (if applicable)



## Secondary char\_type

Encoding	char_type	secondary char type
default	char	none
raw	unsigned char	char (and std::byte)
ASCII	char	none
UTF-8	char8_t	char
UTF-16	char16_t	wchar_t (if applicable)
UTF-32	char32_t	wchar_t (if applicable)

```
auto input = zstring_input<utf8_encoding>("char string");  
const char8_t* ptr = input.begin();
```

Features I'd like to have:

- Determine encoding of char string literals
- Blessing about reinterpret casting between “old” (`wchar_t`, `char`) and “new” (`charXX_t`) char types

Features I'd like to have:

- Determine encoding of char string literals
- Blessing about reinterpret casting between “old” (`wchar_t`, `char`) and “new” (`charXX_t`) char types
- (`char8_t` back in C++98)

Features I'd like to have:

- Determine encoding of char string literals
- Blessing about reinterpret casting between “old” (`wchar_t`, `char`) and “new” (`charXX_t`) char types
- (`char8_t` back in C++98)

Features I'd probably use if they were there:

- `encoding guess_string_encoding(const std::byte* begin, const std::byte* end)`

## Unicode-Aware Rules

```
template <typename Reader>
bool parse_42(Reader& reader)
{
    if (reader.peek() != '4')
        return false;
    reader.bump();

    if (reader.peek() != '2')
        return false;
    reader.bump();

    return true;
}
```

```
template <typename Reader>
bool parse_42(Reader& reader)
{
    using encoding = typename Reader::encoding;

    if (reader.peek() != _char_to_int_type<encoding>('4'))
        return false;
    reader.bump();

    if (reader.peek() != _char_to_int_type<encoding>('2'))
        return false;
    reader.bump();

    return true;
}
```

# Character to int\_type

```
template <typename Encoding, typename CharT>  
constexpr auto _char_to_int_type(CharT c)  
-> typename Encoding::int_type;
```

Uses `Encoding::to_int_type()` with the following requirements:



# Character to int\_type

```
template <typename Encoding, typename CharT>  
constexpr auto _char_to_int_type(CharT c)  
-> typename Encoding::int_type;
```

Uses `Encoding::to_int_type()` with the following requirements:

- if `CharT` is `Encoding::char_type`: everything is allowed

## Character to int\_type

```
template <typename Encoding, typename CharT>  
constexpr auto _char_to_int_type(CharT c)  
-> typename Encoding::int_type;
```

Uses `Encoding::to_int_type()` with the following requirements:

- if `CharT` is `Encoding::char_type`: everything is allowed
- if `CharT` is `unsigned char`: only if `sizeof(Encoding::char_type) == 1`

## Character to int\_type

```
template <typename Encoding, typename CharT>  
constexpr auto _char_to_int_type(CharT c)  
-> typename Encoding::int_type;
```

Uses `Encoding::to_int_type()` with the following requirements:

- if `CharT` is `Encoding::char_type`: everything is allowed
- if `CharT` is `unsigned char`: only if `sizeof(Encoding::char_type) == 1`
- otherwise: `c` must be an ASCII character

lexy only transcodes ASCII → some other encoding.

lexy only transcodes ASCII → some other encoding.

Assumes this is a `static_cast<typename Encoding::char_type>(c)`.

## LEXY\_LIT() rule

```
static constexpr auto rule = LEXY_LIT("Hello World!");
```

## LEXY\_LIT() rule

```
static constexpr auto rule = LEXY_LIT("Hello World!");
```

Transcoding:

- LEXY\_LIT("a"): always ok, ASCII character

## LEXY\_LIT() rule

```
static constexpr auto rule = LEXY_LIT("Hello World!");
```

Transcoding:

- LEXY\_LIT("a"): always ok, ASCII character
- LEXY\_LIT("ä"): requires `Encoding::char_type == char`



## LEXY\_LIT() rule

```
static constexpr auto rule = LEXY_LIT("Hello World!");
```

Transcoding:

- LEXY\_LIT("a"): always ok, ASCII character
- LEXY\_LIT("ä"): requires `Encoding::char_type == char`
- LEXY\_LIT(u8"a"): always ok, ASCII character

## LEXY\_LIT() rule

```
static constexpr auto rule = LEXY_LIT("Hello World!");
```

Transcoding:

- LEXY\_LIT("a"): always ok, ASCII character
- LEXY\_LIT("ä"): requires `Encoding::char_type == char`
- LEXY\_LIT(u8"a"): always ok, ASCII character
- LEXY\_LIT(u8"ä"): requires `Encoding::char_type == char8_t`

## LEXY\_LIT() rule

```
static constexpr auto rule = LEXY_LIT("Hello World!");
```

Transcoding:

- LEXY\_LIT("a"): always ok, ASCII character
- LEXY\_LIT("ä"): requires `Encoding::char_type == char`
- LEXY\_LIT(u8"a"): always ok, ASCII character
- LEXY\_LIT(u8"ä"): requires `Encoding::char_type == char8_t`
- LEXY\_LIT(L"ä"): no pre-defined `Encoding` matches it

```
// Matches `0xFF`, `0xFE`.  
static constexpr auto rule  
    = dsl::bom<lexy::utf16_encoding, lexy::encoding_endianness::little>;
```

```
// Matches `0xFF`, `0xFE`.
```

```
static constexpr auto rule
```

```
= dsl::bom<lexy::utf16_encoding, lexy::encoding_endianness::little>;
```

- UTF-16/UTF-32 BOM in little/big endian
- UTF-8 BOM (endianness ignored)
- for all other encodings: no-op

# Code point rule

```
// Matches a single arbitrary code points in the input encoding.  
static constexpr auto rule = dsl::code_point;
```

- ASCII: consumes one character
- UTF-8/16/32: one Unicode code point
- default/raw encoding: compile-time error

lexy does not validate that the input uses the specified encoding.

lexy does not validate that the input uses the specified encoding.

- `LEXY_LIT(u8"...")`: just matches that exact same byte sequence
- `dsl::until(LEXY_LIT("\n"))`: skips code units until `\n` is found
- `dsl::code_point`: fails if code units don't form a valid code point



lexy does not validate that the input uses the specified encoding.

- `LEXY_LIT(u8"...")`: just matches that exact same byte sequence
- `dsl::until(LEXY_LIT("\n"))`: skips code units until `\n` is found
- `dsl::code_point`: fails if code units don't form a valid code point

Unexpected early EOF if input contains the sentinel value.

## Other rules worth mentioning

- `dsl::newline: \n or \r\n`
- `dsl::ascii::*: e.g. dsl::ascii::alpha`
- `dsl::digit<Base>: one of the ASCII digits`

Features I'd like to have:

- Unicode equivalents to `cctype`

Features I'd like to have:

- Unicode equivalents to `cctype`

Features I'd probably use if they were there:

- `std::code_point`
- encoding validation

Features I'd like to have:

- Unicode equivalents to `cctype`

Features I'd probably use if they were there:

- `std::code_point`
- encoding validation

Features I don't actually need:

- `std::parse_code_point()`
- transcoding facilities

# File Input

```
template <typename Encoding>
class buffer
{
public:
    using encoding = Encoding;
    using char_type = typename Encoding::char_type;

    explicit buffer(const char_type* data, std::size_t size);
    template <typename SecondaryCharT>
    explicit buffer(const SecondaryCharT* data, std::size_t size);

    auto reader() const&;
};
```

```
template <typename Encoding, encoding_endianness Endianness>  
buffer<Encoding> make_buffer(const void* memory, std::size_t size);
```

- create `Encoding::char_type` from raw bytes
- handle endianness conversion



```
enum class encoding_endianness  
{  
    little,  
    big,  
    bom,  
};
```

```
enum class encoding_endianness
{
    little,
    big,
    bom,
};
```

If bom is used:

- check for little/big BOM of the Encoding and act appropriately
- if no BOM present: assume big endian
- BOM removed from the input

# Encode rule

```
static constexpr auto rule = []{  
    auto cp = dsl::code_point;  
  
    // Temporarily assume a different encoding.  
    auto first  = dsl::encode<utf8_encoding>(cp);  
    auto second = dsl::encode<utf16_encoding>(cp);  
    auto third  
        = dsl::encode<utf16_encoding, encoding_endianness::little>(cp);  
  
    return first + second + third;  
}();
```

No transcoding: requires raw\_encoding as the input.

```
template <typename Encoding = default_encoding,  
         typename Endianness = encoding_endianness::bom>  
buffer<Encoding> read_file(const char* path); // modulo error handling
```

- no newline conversion (binary mode)
- no encoding validation
- appropriate endianness handling

Features I'd like to have:

- `std::read_file()`

Features I'd like to have:

- `std::read_file()`

Features I'd probably use if they were there:

- endian conversion function
- `encoding_guess_string_encoding(const std::byte* begin, const std::byte* end)`

## The `as_string` callback

# dsl::capture()

```
struct production
{
    // 123456789
    static constexpr auto rule = dsl::capture(dsl::digits<>);
    static constexpr auto value
        = lexy::callback([](auto lexeme) { ... });
};
```



```
template <typename Reader>
class lexeme
{
public:
    iterator begin() const;
    iterator end() const;

    std::size_t size() const;
};
```

```
struct production
{
    // 123456789
    static constexpr auto rule = dsl::capture(dsl::digits<>);
    static constexpr auto value
        = lexy::as_string<std::string>;
};
```

```
template <typename String>
struct _as_string
{
    String operator()(String&&) const;
    String operator()(const /* CharT */ *str, std::size_t length) const;

    template <typename Reader>
    String operator()(lexeme<Reader> lex) const;
};

template <typename String>
constexpr auto as_string = _as_string<String>{};
```

```
template <typename String>
struct _as_string
{
    String operator()(String&&) const;
    String operator()(const /* CharT */ *str, std::size_t length) const;

    template <typename Reader>
    String operator()(lexeme<Reader> lex) const;
};

template <typename String>
constexpr auto as_string = _as_string<String>{};
```

No transcoding: String::value\_type must match primary or secondary  
Encoding::char\_type.

## dsl::quoted()

```
struct production
{
    // "string literal"
    static constexpr auto rule
        = dsl::quoted(dsl::code_point - dsl::ascii::control);
    static constexpr auto value
        = lexy::as_string<std::string>;
};
```

## dsl::quoted()

```
struct production
{
    // "string literal"
    static constexpr auto rule
        = dsl::quoted(dsl::code_point - dsl::ascii::control);
    static constexpr auto value
        = lexy::as_string<std::string>;
};
```

How should the code point be encoded?

```
template <typename String, typename Encoding>
struct _as_string
{
    ...

    String operator()(lexy::code_point cp) const;
};
```

```
template <typename String, typename Encoding = /* deduce from char type
constexpr auto as_string = _as_string<String, Encoding>{};
```

Features I'd probably use if they were there:

- code point to string conversion
- transcoding functions